MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Experimenting with the Robot Operating System (ROS) to enable communication between drones in the Dronology project.

Eryoruk, Huseyin

Award date:
2020

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Experimenting with the Robot Operating System (ROS) to enable communication between drones in the Dronology project.

Eryoruk Huseyin

RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Preface

This dissertation has been written after my internship at the University of Notre Dame, Indiana. This is the conclusion of two years of studying within the University of Namur in Computer Science. It is thus the last step before getting my Master's degree. The overall context of my visit was about unmanned aerial vehicle, which will be the main subject of this work. It addresses the different challenges that I have faced while searching and working with autonomous drone flight. My stay at Notre Dame helped me discover the universe of drones, both on software and on hardware, with whom I have worked for three months. I have learned the terms and technology, such as Robotic Operating System and Dronekit, two frameworks used to develop autonomous solutions for drones.

I had several reasons that motivated me to go to the University of Notre-Dame. The first one was the experience of going abroad, where I can improve my autonomy. And to improve my ability of solving problems, a skill that I have developed throughout my studies at the University of Namur, Belgium. My second motivation was the subject, which was an important factor in my decision. The drone universe is going to be the next challenge that the society will have to face, in many fields. The society will have to respond to the issues on both ethical and integration level. The third reason is that visiting another country means learning the culture, the way people live, meet new people with different views, and most importantly to talk the local language, going to the USA was a good way to improve my English. Even if we have learned the language through courses in school, the best way to enhance and consolidate it remains the daily practice. And the last reason would be the challenge. I wanted to challenge myself, by going abroad, where I have never been. I believe that by getting out of his comfort zone, a person can grow mentally, physically and acquire new skills.

I would like to thank Prof. Heymans for the chance and trust that he gave me by giving me the opportunity to participate in this project, and the help and advice that he gave me for during the writing process. I also would like to thank Prof. Huang for letting me be part of the Dronology project and her help during my stay. I am also thankful to Dr. Vierhauser for being available and his help during my stay.

I hope that you will enjoy reading my work as much as I enjoyed writing it.
Eryoruk Huseyin.

Abstract

Unmanned Aerial Vehicles, more commonly known as UAVs, are becoming increasingly popular. In the past decade tremendous improvements have been made in every aspect of the domain, such as software solutions, hardware, design, etc. UAVs were being used for military goals first, before starting to be used for recreational and for commercial purposes.
The needs are evolving alongside with the technologies. The current solutions are mostly focused on flying single drones. As the needs are becoming more complex, the necessity of flying several drones is increasing. Developed by the University of Notre Dame, Dronology is addressing this need. Dronology is a software for drone management and flight coordination using Dronekit. It allows to create flights and deploy multiple UAVs. All the commands are sent from the ground and the drones do not communicate with each other, a limitation due to Dronekit. Overcoming the latter is very important as it will make it possible to develop complex systems for monitoring, search-and-rescue, etc.
This thesis tries to address this limitation by using a technology different from Dronekit, called Robotic Operating System. The two main questions addresses in this work are : (1) Is ROS suitable for drone. (2) How to achieve data exchange between flying drones?

Keywords: UAV, drone, software, Dronekit, robotic operating system.

Table of content

Chapter 1 Introduction

1.1 Context

Unmanned aerial vehicles (UAV), more frequently known as drones, have become increasingly popular. In recent years, they are being used for both research and recreational purposes. With the significant progress made in both software and hardware, they started to get solicited for professional tasks too. Like in the '60s, when robots started to get used in the manufacturing industry. Today drones have become mature enough to be used to offer new services or to improve the efficiency of humans on certain tasks.
For example, they are deployed to support search tasks, surveillance, mapping, 3D modelling, data collection and many other tasks. One recent usage example was the fire at Notre-Dame Paris: firefighters deployed drones to detect and assess damages on the cathedral's roof. Another example is Amazon, Uber and their new delivery programs. All these sudden interest are signs that drones are becoming widespread and in the coming decades will be part of our growing society in response to new needs coming along.

In order to make it possible, research needs to be carried on. Universities, research institutes and corporations have to do research about UAVs to discover new use cases, to improve the technology and to control it. Currently, new use cases are being proposed every day and new solutions to achieve them are being explored, often involving tradeoffs between concerns such as performance, safety and efficiency.
My stay at University of Notre Dame (South Bend, IN) was about UAVs and semi-autonomous flights of multiple aircraft sharing the same airspace.
To efficiently assist humans, drones need to have a certain level of autonomy and intelligence. They have to be able to fly and operate towards a location with minimal human interaction. Which makes them autonomous but not completely as human assistance will always be solicited.

My main focus was to explore and improve by using proof of concept, the drone technology used in the Dronology software. Dronology's first goal is to be an incubator to study different aspects of software development for cyber-physical systems such as drones. It is a place where researchers can test and verify their solutions within a working environment.
One of the recent research, from the University, was focusing on solving a routing problem. The overall context was the rescue of a missing person in a river by using drones. Drones would be deployed by swarms to find the person. A river can be a big search area, which involves an infinite number of possible routes to explore for each drone. Thus the deployment needs to be planned and configured in a certain way so that the objective is accomplished with the most efficiency possible. Time constraint is very important when it comes to human lives, especially when a person is in danger of drowning.

This is a time-critical event thus the efficiency is directly linked to it. In order to go through all the possibilities of the search space, the research team came up with a solution using a genetic algorithm and a goal model. The solution is generating routes for each drone that will allow finding the individual quickly, considering the different goals that need to be achieved.
Dronology also provides interfaces for users to control, plan and coordinate drone flights, both single and multiple drones. The latter is very important because many of the current solutions in the industry are focused on a single flight whereas Dronology is focusing on coordinating several UAVs that collaborate to achieve a complex task. Dronology is developed with a framework called Dronekit. It is an open-source and community-driven tool launched in 2015, for developers to create software for UAVs. It is meant to create software to run on computers to interact with UAVs. It is providing necessary features and tools with SDK and an API to develop drone software.

1.2 Objective

Although Dronekit facilitates drone software development, it has nevertheless some important limitations. There are two limitations that are very important. First, the latency that occurs due to the fact that the software is running on the computer instead of being in the actual drone. This drawback, however, will not be addressed in this document. And second, Dronekit doesn't allow communication in the air between drones. This aspect is very important and will be the main subject of the contribution chapter.
The ability to communicate in the air could open a lot more new perspective for solutions. It will first reduce the dependency to the ground and increase the autonomy of the drones. But also bring a new aspect, the sharing of data in the air.
Therefore, we could take the example above and imagine a solution where the routing algorithm runs in the air on a specific drone with the result being transmitted to the other members of the group. There are several reasons to look for alternatives to Dronekit. The first reason is that currently only Dronekit has been explored as a solution for drone development. Therefore, all the future solutions will have to be designed and decided according to the current technology. Which will make Dronology be stuck to Dronekit possibilities and limitations. The second reason is related to the previous one, the need of evolution. Dronology in his current state is mature enough to start thinking about new functionalities. The current technology doesn't provide the necessary support to make UAVs exchange information during their flights. This functionality allows drones to collect, share data and take decisions during their mission.
The research and development work that I achieved was aimed at addressing these limitations by considering alternative technologies and testing them. More specifically, my main work was focused on testing another framework called Robotic Operating System (ROS).

In ROS, the terms "operating system" may be misleading as ROS, just like Dronekit, provides a set of tools and paradigm to develop software. However, ROS and Dronekit are different in several ways. Their first difference is their respective main target: Dronekit focuses on UAVs only while ROS focuses on robotics. Also, ROS is not meant to be the best framework with the richest set of features. Instead it is providing OS level features such as messages passing between processes, hardware abstraction etc.

My work at Notre Dame was to address these limitations by using ROS. I started by learning ROS, its concepts and its underlying paradigm. Then I had to discover how drone flights can be implemented in ROS which is substantially different from Dronekit. When I started to master ROS, I implemented all the necessary functions and methods to make a drone successfully fly. All the code have been tested in a simulator. A simulator is an environment with whom user can create outdoor conditions for drones and allow them to safely test their solutions without hardware. Instead of controlling actual drones, the software will interact with generated UAVs and allow users to test the behavior of their solutions. Flight data will be analyzed to ensure the validity of the solutions.
Once the behavior inside the simulator matched the expectations, we proceeded to real life tests (unfortunately not all outdoor tests could be made due to weather and time).
Outdoor testing was conducted on new hardware which had to be setup and prepared to fly. Afterward, it involved ROS's installation and configuration within the drone. So these tests were meant to assess both the hardware first and then the software.

The main contribution of this Master thesis is the exploration of ROS as an alternative to Dronekit. First, by implementing codes to make drones fly and ensure that it's working properly. Second, finding a way to use ROS so that information can be exchanged between drones inside the simulator. Third, configure the drones so that ROS can be run on them and test the solutions on real hardware.

1.3 Structure

This rest of this document is split in three chapter. The first chapter, State of the art, will contain all the technology that I've worked with. The first section introduces the term Unmanned Aerial Vehicle as well its component. Followed a presentation of Dronology and his architecture. Following by other components and concepts such as ROS. The different protocol to establish communication, the software environment of tests. The concept of Autopilot and flight controllers.
The next chapter, Contribution, will contain my work, both hardware and software. The first section will present the different modifications that I've brought to the physical drone to

ensure that ROS runs correctly as well as an explanation of the architecture of the components.
Then, the next chapter will illustrate the ROS implementation and observations that occurred while testing the solutions.
Finally, the last chapter will contain the conclusion of this document.

Chapter 2 State of the art

2.1 Unmanned Aerial Vehicle

2.1.1 Definition and History

The term Unmanned Aerial Vehicle (UAV), also known as drone, is given to a vehicle that does not need pilots to be controlled. Instead the control is done by a remote station.

In 1999, P. Van Blyenburgh has defined the UAV as "uninhabited and reusable motorized aerial vehicles, which are remotely controlled, semi-autonomous, autonomous, or have a combination of these capabilities[...]" [1]. Later in 2017 the definition has not much changed as the British Ministry of Defense has defined Unmanned Aircraft (UA) [2] as " An aircraft that does not carry a human operator, is operated remotely using varying levels of automated functions, is normally recoverable, and can carry a lethal or non-lethal payload."

And the Unmanned Aerial System (UAS) [2] as " A system, whose components include the unmanned aircraft and all equipment, network and personnel necessary to control the unmanned aircraft." The North Atlantic Treaty Organization (NATO) shares the same definitions [2]. UAS will be used to represent the whole system : the vehicle, the control system, and the pilot. While the UAV refers to the flying vehicle. The term UAV has evolved and changed over the past years as adoption is growing. Other terms are used to refer to UAV like Remotely Piloted Aircraft (RPA) or Remotely Piloted Aircraft System (RPAS) and other terms cited by Francesco Nex and Fabio Remondino [3] and also by Kimon P. Valavanis and George J. Vahctevanos [4]. In the literature, the terms UAV and UAS are often interchangeable and have been commonly used to refer to UA.

History

The research and development first started after the First World War around 1916, developed by the American Military [5] [6] [7] [8]. The first functional UAV has come to success in late 1917 developed in a secret lab run by Orville Wright and Harles F Kettering. They designed the first UAV, in Figure 1, called Kettering Bug, capable of carrying a 180 pounds (81,1 kg) bomb with a 40-horsepower engine manufactured by Ford. Formerly called Kettering Aerial Torpedo, the Kettering Bug had a distance range that was above any ground artillery and was designed to carry bombs. The Bug was just a precursor then of what cruise missiles are today. To reach its target, Kettering and Wright had the idea to calculate the number of revolutions necessary to carry Bug to its target using three values the wind speed, the wind direction and the distance to target. Using these parameters, the value was calculated and set, after N revolution the cam dropped shutting off the engine and releasing the wings from the fuselage which makes the body drop instantly. The vehicle has never been tested on the field because the war ended soon before the first UAV was fully developed and functional. Two other UAVs have been developed before but they were concepts and have never reached the production step.

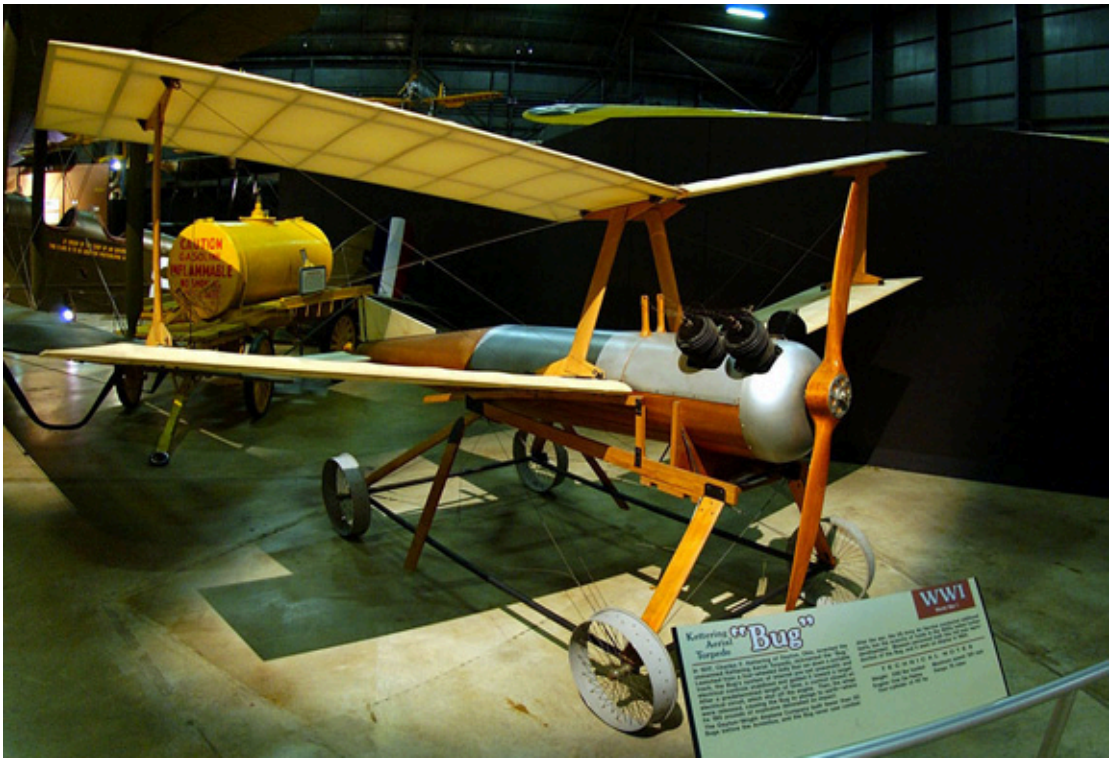

Figure 1 - The Kettering "Bug" Source: National Museum of the United States Air Force.

The GPS, the Internet, digital photography are technologies that were launched initially for military usage that became later used by civilians. Years after the WWI, despite being heavily used for military, drones are now starting to get used for civilian applications. Over the decade, they became more sophisticated, more efficient as studies have been conducted for design, systems, proof of concept and usage. Even though they are not yet mature enough to operate safely in urban airspace, UAVs are becoming widely used for civilian tasks as they are much cheaper and convenient than manned aerial vehicles and much lower risk to human life.

2.1.2 Classification

UAVs have become increasingly sophisticated to respond to social, business and war needs. With the amount of variation for different usages, a classification and a general framework have become more than needed. Classification is required to provide a common view and therefore help the communication between parties on different scales. Several classifications can be found. Each of them using different parameters to create categories: the weight [9], distance range [10], type and aerodynamics [11] and even based on the landing [11].

2.1.2.1 Military classification using weight and altitude range

The weight of an aircraft is defined by many parameters. Therefore a category can be created for each parameter. The weight used by NATO is defined by the maximum gross weight at take-off (MGTOW) [9]. The MGTOW of an aircraft is defined [12] [13] as the weight of an aircraft that the pilot is allowed to carry at take-off. It is a particular definition of the general definition of an aircraft gross weight [12] that is defined as the weight of an aircraft at any moment during flight, illustrated in Figure 2.

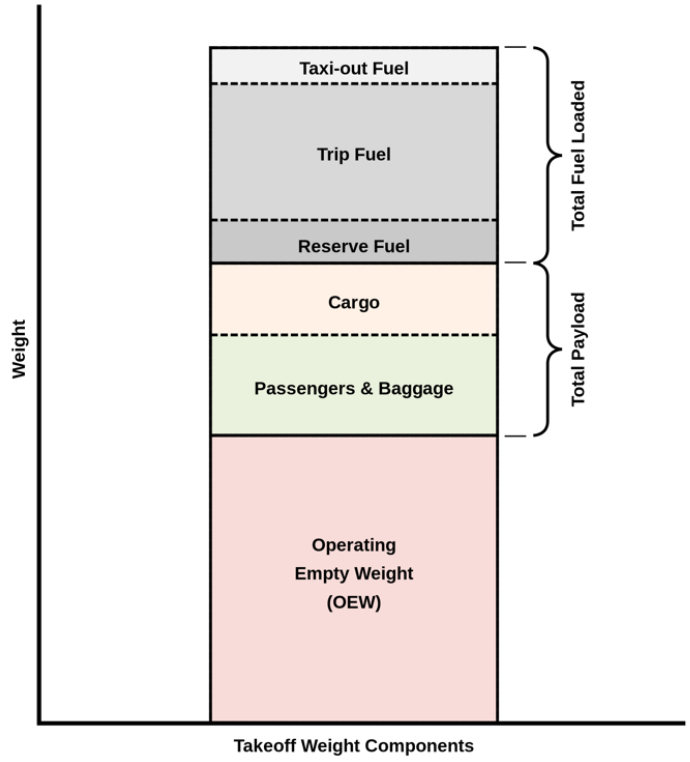

Figure 2 - The composition of the Take-off Weight of an Airplane.

The reason why the take-off is used for classification is simple. In the case of a commercial flight, during the flight, the weight doesn't vary much as anything is added or removed from the aircraft except the fuel, the oil, etc. However, it is not the case with the UAVs as they can carry weights, like a lethal weapon, that could be dropped and significantly reduce the weight. The weight and the normal operating altitude are two simple classifiers that are easy to understand as they are self-explanatory. A unified classification, that defines classes and subclasses, has been defined by NATO based on these two parameters. The classes are defined by the weight, followed by the operational altitude that defines the subclasses. These categories provide a unified language for countries when it comes to UAV. Table 1 is an illustration of the categories defined by NATO and Figure 3 - Classification repartition of UAVs shows the repartition of UAVs according to their size and altitude capacity.

Class	Category	Normal employment	Normal Operating Altitude	Normal Mission Radius	Primary Supported Commander	Example platform
CLASS I (less than 150 kg)	SMALL >20 kg	Tactical Unit (employs launch system)	Up to 5K ft AGL	50 km (LOS)	BN/Regt, BG	Luna, Hermes 90
	MINI 2-20 kg	Tactical Sub-unit (manual launch)	Up to 3K ft AGL	25 km (LOS)	Coy/Sqn	Scan Eagle, Skylark, Raven, DH3, Aladin, Strix
	MICRO <2 kg	Tactical PI, Sect, Individual (single operator)	Up to 200 ft AGL	5 km (LOS)	PI, Sect	Black Widow
CLASS II (150 kg to 600 kg)	TACTICAL	Tactical Formation	Up to 10,000 ft AGL	200 km (LOS)	Bde Comd	Sperwer, Iview 250, Hermes 450, Aerostar, Ranger
CLASS III (more than 600 kg)	Strike/ Combat	Strategic/National	Up to 65,000 ft	Unlimited (BLOS)	Theatre COM	
	HALE	Strategic/National	Up to 65,000 ft	Unlimited (BLOS)	Theatre COM	Global Hawk
	MALE	Operational/Theatre	Up to 45,000 ft MSL	Unlimited (BLOS)	JTF COM	Predator B, Predator A, Heron, Heron TP, Hermes 900

Table 1 - UAV classification provided by NATO. [2]

Figure 4 and Figure 5 are examples of Class I and II. The left UAV in Figure 4 is a Small category UAV called Hermes 90, it weighs 110kg. On the right side is a mini UAV example, called Aladin. It weights 4kg. Figure 5 contains two examples of the same class, Class II. Hermes 450 and Sperwer. The former weights 600 kg and the latter weights 330 kg. Finally, an example of Class III is illustrated by Figure 6, the Global Hawk with a weight of 14 628 kg. All the weights are expressing the MGTOW.

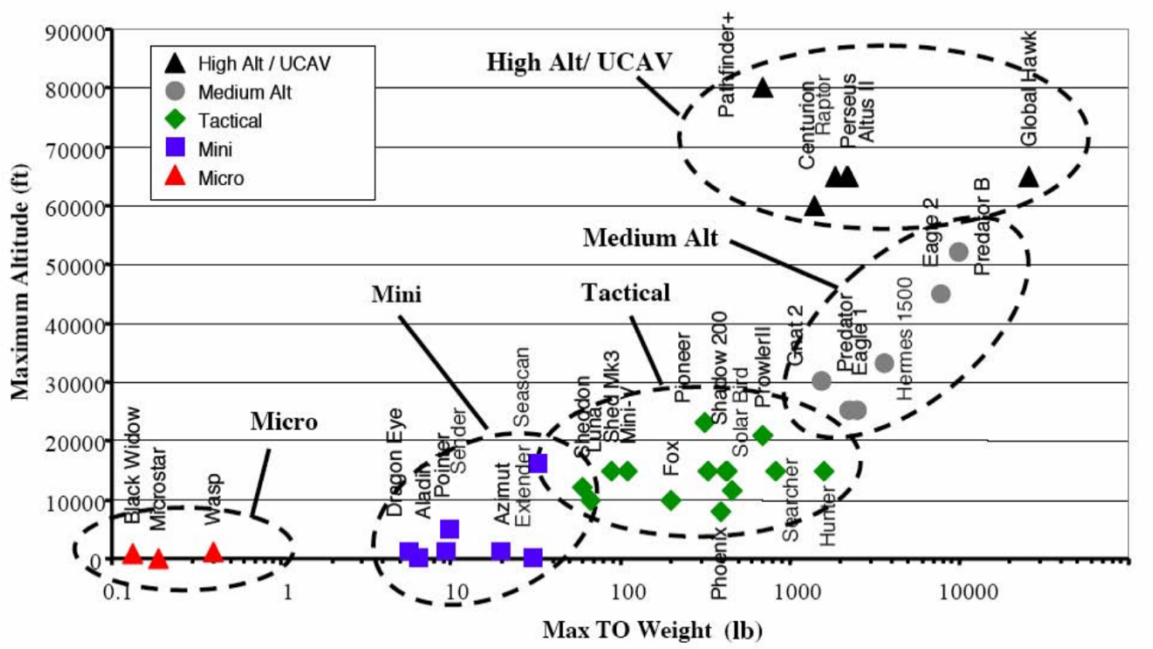

Figure 3 - Classification repartition of UAVs [14]

Figure 4 - Two examples of Class I UAVs, Small and Mini. [15] [16]

Figure 5 - Two examples of Class II UAVs. [17] & By David Monniaux

Figure 6 - Class III UAV, Global Hawk. [18]

2.1.2.2 Morphology/Type

Over the past years, a lot of flying aircraft types have been developed, to reply to the constant evolution of the needs of society. Systems have been developed around different types of UAVs and have been used for different applications, fixed wings for surveillance, mapping and payload dropping [3] [19], multi-copters to support Wireless Sensor Network [20], VTOL to help law enforcement [21], etc.

A more general classification (compared to classification in section 2.1.2.1) is defined using the shape of the vehicle. This classification will be easier to recognize as the category of the aircraft can be determined without any quantitative value. Opposite to weight classification where it can be difficult to identify the class of an aircraft.

2.1.2.2.1 Fixed Wings

Fixed-wing UAV is characterized by static two wings, looking more like the traditional manned aircraft. A fixed-wing has its own advantage, as explained by Eric N. Johnson [22], like the take-off using human hands.

They are known to have better stability and better handling of weather conditions. Which makes the fixed-wing UAV more suitable for surveillance as the range is bigger than multi-copter aircraft due to their size and their stability [22] [23]. They are currently used by the military for scanning war zones quickly and safely (no potential of human lost) but also for agriculture and environmental industries.

They can also carry a greater payload and use less power than a multi-copter vehicle due to the airfoil provided by the wings. Moreover, a fixed-wing aircraft has a less complex structure

as it is composed of one body and two wings as opposed to multi-copter UAV that has several arms and motors. But also, they are less prone to collateral damage in case of motor defects, power loss, or battery issues as fixed-wing UAV can slowly drift and make a safer landing. In opposite, a multi-copter drone would crash much faster if one of the motors would have to stop, leading to more damage.

2.1.2.2.2 Multi-Copter

Multi-rotor or multi-copter UAV is defined by multiple arms (numbers can vary) carrying a rotor. It has several advantages compared to a fixed-wing vehicle [22]. First, the multicopter does not need a long place to land or take-off as it can land at a specific point which is a big advantage for urban environment or forest, etc. It has much faster speed and movement freedom. It does not need the same amount of space to exercise a move. However, the main drawback is that multicopter drones do not have an airfoil design therefore they have to create their own lift, resulting in much bigger power consumption and reduced endurance.

2.1.3 Yaw, Roll, Pitch

Aircrafts are controlled and guided by three aeronautical concepts. The yaw, roll and pitch, are used to describe the motions responsible for the orientation and rotation movement of an aircraft through the three axes (x,y,z) of the three-dimensional space [24] [25]. They represent the principal axes of the Euclidean space displayed in the Figure 7 – Representation of a point in the Euclidean space.

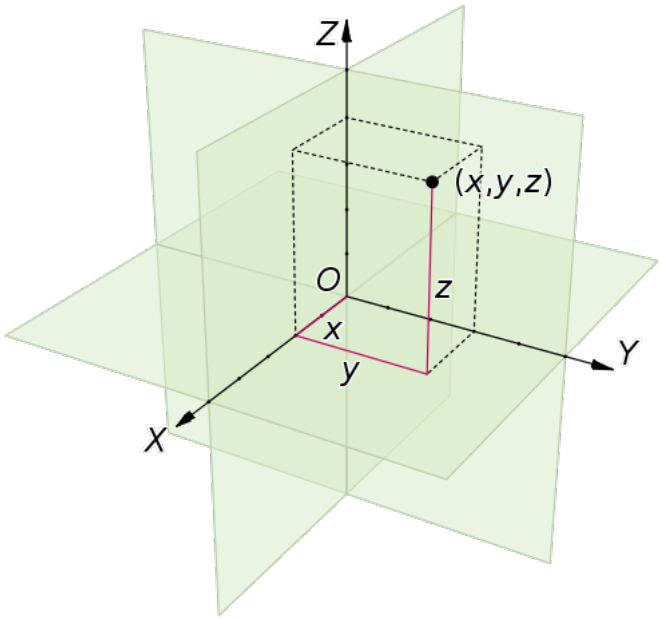

Figure 7 – Representation of a point in the Euclidean space.

The three concepts are also a derivative of the Euler angles, called Trait-Bryant angles, used to describe the **orientation** of a body based on the fixed coordinate. The orientation of body is called attitude.

The Tait-Bryan angles are a formalism called after Peter G. Trait and George H. Bryant, used for aircraft applications. Each of these axes goes through the center of gravity of the aircraft, around which aircraft rotates.

The **yaw axis** represents the vertical axis, Z-axis , perpendicular to the fuselage of the plane it goes to the bottom of the fuselage and will create what is called the yaw. The yaw motion is created when a positive force is applied to the axis it will make the nose, or the heading, of the plane turn to right. A negative force will make the nose turn left. The movement is illustrated by the sketch in Figure 8, the heading Rotation. When the down axis is turning right, the nose of the aircraft is also turning right.

The **roll axis** represents the longitudinal axis, X-axis, parallel to the fuselage from the nose of the plan to the tail. The roll axis is responsible for the roll motion, which is the leaning of the plane to one side. A roll motion will lean one wing down while the other one goes up. It results in a movement to the right or to the left. The movement is illustrated on the last part of Figure 8. The transverse axis is rotating to the right which results in an elevation of the nose upwards, the variation is shown by the red arrow.

Finally, the **pitch axis** represents the lateral axis, Y-axis, parallel to the wings from one end to the other. The axis always points to the right wing. The motion is called the pitch. A positive pitch motion will raise the nose up and lower the tail, a negative pitch will have the opposite effect.

The movement is illustrated by the pitch Rotation 3 in Figure 8. This time the longitudinal axis is rotating to the right which makes the right wing tip head down while the opposite wing is heading up.

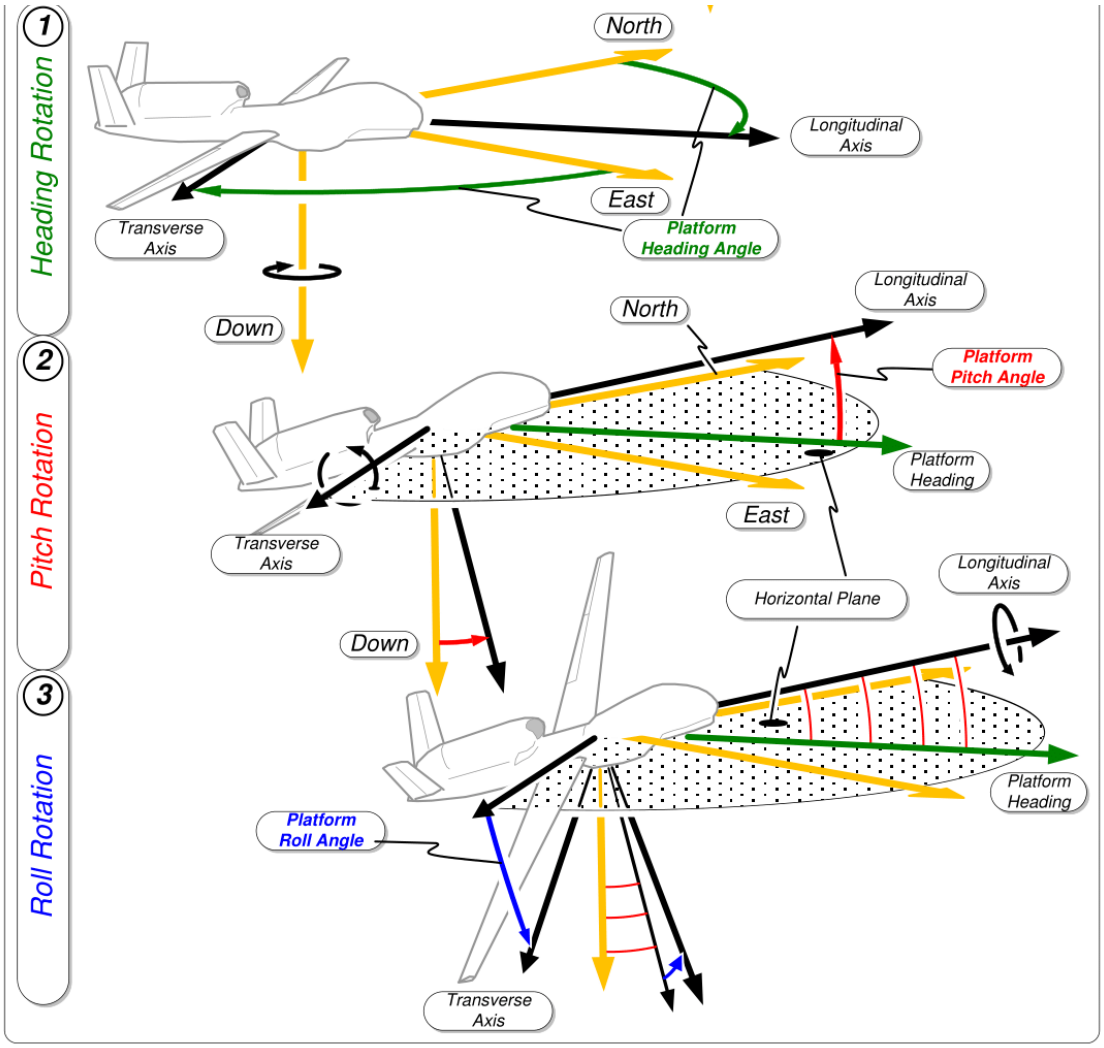

Figure 8 - Yaw, Pitch, Roll description on an aircraft. [26]

2.1.4 Telemetry and Radio Command

Telemetry is the collection and transmission of data from a remote source to an IT system for monitoring [27]. These data are collected from different types of sensors that take different measurements. The real-time data transmission can be done with different methods like radio, ultrasonic, infrared, GSM, depending on the application. Telemetry is used in many disciplines such as weather measurement and forecasting [28], agriculture monitoring, energy monitoring, and even marine animal surveillance [29]. In the context of UAV, Radio signals will be used for Telemetry. The data are collected from different onboard sensors, like a gyroscope, GPS, and are sent to the companion computer. It is a two way data stream where information such as battery status, GPS data for positioning is exchanged. The list does not stop here. One can get way more information about the aircraft, like the speed, the altitude, the rpm of the propellers, etc. Telemetry frequency varies by country and hardware. The Radio

frequency spectrum is a limited resource thus a good usage is mandatory. Each country has its own frequency allocation.
In 1992, Europe has decided to adopt a common European Table of Frequency Allocations And Utilisations [30] to harmonize the frequency usage within the territory.
For instance, radio frequency for Telemetry in Europe is 433 MHz [31] while 915MHz [32] is used in the USA. Those values are part of the license-free Industrial, Scientific, Medical (ISM) band. The ISM band is defined by the International Telecommunication Union (ITU) Radio Regulation [33].

While Telemetry is used to retrieve and exchange data, radio control (R/C or RC) is used to remotely control a device. A radio control system is composed of two elements, a transmitter and a receiver. Radio signals are sent from a transmitter (the controller) to a receiver (the controlled device) via a set of radio frequencies to give control commands. These frequencies are usually 2.4GHz and 5.0GHz depending on the protocol and hardware. Both are also the frequencies of the WIFI, which can lead to instability and connection losses in urban areas.
R/C is a one way data stream that the receiver does not reply to but only interprets the commands. However, an RC controller can have several channels of data transmission, each channel corresponding to one information. The information is mainly to control the aircraft and manipulate the yaw, pitch and roll, described in section 2.1.3. Each of these information can be transmitted by an individual channel depending on the protocol. There are few radio control protocols that are available for both sides, transmitter (TX) and receiver (RX). Each of them tries to improve the existing solutions and to solve fundamental communication problems like integrity, confidentiality, reliability (e.g. issues like interference, collisions of packets, packet recovery).
While Telemetry is a way of exchanging information, it needs to come with a messaging protocol to ensure the communication.

2.1.5 MAVLink

Micro Air Vehicle communication protocol, MAVLink, is an open-source protocol, under the LGPL license created in 2009, used to carry information and to establish communication with UAVs [34]. It is a lightweight messaging protocol using a publish-subscribe pattern and point-to-point design pattern. It is designed for restricted bandwidth and resource-limited systems like UAVs. It is available for many programming languages which make it usable on many microcontrollers and operating systems (MacOS, Linux, Microsoft Windows OS). It has two versions: V1.0 and V2.0. MAVLink 2 is an update of the protocol to bring more security and flexibility. It implements packet signing, has a bigger range for message ID, removes the empty byte at the end of the payload, etc.

Messages

Messages are defined via XML files, each of these files defines a particular MAVLink system message. Figure 10 is showing a list of those files. Each of these files define a set of message types and corresponding values. Each of these messages is defined with an Enumeration type shown in Figure 9. These sets of messages are supported by almost all the ground control stations and autopilots. The definitions cover the necessary functionality for the autopilot and the ground station software. There are two categories of messages: command and state. The command messages are usually sent by the Ground station software to execute actions. The state messages are sent from the UAV to the ground containing information such as battery state, altitude, etc. Other examples are: the autopilot used, the type of aircraft, failure flags, etc.

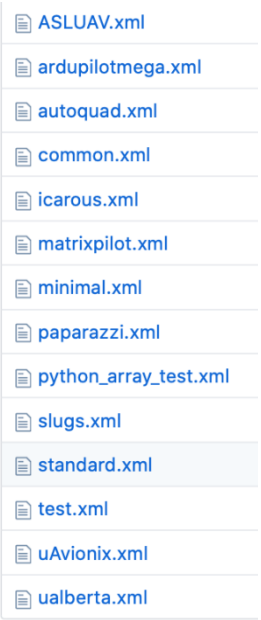

Figure 10 - XML file list provided by MAVlink

```
<enum name="HL_FAILURE_FLAG">
  <description>Flags to report failure cases over the high latency telemtry.</description>
  <entry value="1" name="HL_FAILURE_FLAG_GPS">
    <description>GPS failure.</description>
  </entry>
  <entry value="2" name="HL_FAILURE_FLAG_DIFFERENTIAL_PRESSURE">
    <description>Differential pressure sensor failure.</description>
  </entry>
  <entry value="4" name="HL_FAILURE_FLAG_ABSOLUTE_PRESSURE">
    <description>Absolute pressure sensor failure.</description>
  </entry>
  <entry value="8" name="HL_FAILURE_FLAG_3D_ACCEL">
    <description>Accelerometer sensor failure.</description>
  </entry>
  <entry value="16" name="HL_FAILURE_FLAG_3D_GYRO">
    <description>Gyroscope sensor failure.</description>
```

Figure 9 - An example of a MAVLink message definition.

Features

MAVLink is meant to be light and secure due to the need for a system to respond as quickly as possible. Integrity is also very important since UAVs are considered as safety-critical. Unauthorized modification of a message by a malicious individual could lead to potential threats. A MAVLink packet has only 8 bytes for the V1.0 and 11 bytes for V2.0. This makes it easy to process and suitable for low bandwidth systems. Those numbers correspond to the smallest packet possible (i.e. without payload). It can go up to 263 bytes for the V1.0 and up to 279 bytes for the V2.0. Figure 11 illustrates the architecture of a packet. MAVLink also

provides two integrity checks to make the delivery reliable against packet loss, corruptions and provide packet authentication.

It integrates the X.25 checksum a Cyclic Redundancy Check (CRC) to ensure the integrity of a message during the transmission that the data has not been altered. And the second verification ensures the integrity of the data type transmitted to make sure that two messages contain the same information. MAVLink supports up to 255 components inside the network.

Usually, a UAV has to deliver information about its state not only to one but to many sources, MAVLink supports multicast streams.

Different modes of communications are available: **point-to-point** and **publish-subscribe**.

The former ensures the delivery of messages to a specific target using the target ID and target component. And the latter will save bandwidth by omitting the target and component ID since no verification is done on the IDs. Generally this mode is used with the multicast mode.

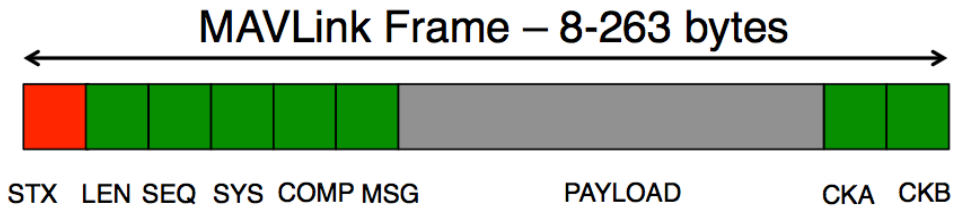

Figure 11 – MAVlink V1 packet illustration.

2.1.6 Flight flow

Figure 12 shows the state diagram of a flight using Dronology. It shows every step of a flight and the different states that the drone is into and the transitions between them.
The blue statement on the graph are features, these are:

- Safety checks
- Mission mode
- Geofence
- Collision Avoidance

Safety check is the process where some parameters of the drones are checked for safety matters. The checks are done to prevent the vehicle from arming with small or large issue such as a low level of battery or bad calibration health. It is a feature provided by the Autopilot that can be disabled. The reader can find a list of all the checks performed in Table 2 - Pre-arm safety checks.
Some examples of verification are:

- Battery level is above a certain level
- GPS signal is above a certain value
- Gyroscope health or calibration.
- Drone is paired with a radio command.

Mission mode is described in section 2.2.3.5.
Collision avoidance is a feature that allows the drone to avoid collision with obstacles. For instance, if a waypoint is given and there is a tree in the path, the drone will detect it and stop or get around it. But this depends on the type of collision avoidance.
Geofence is a safety measure that allows users to set up a virtual fence. It will act like a territory and set the boundaries of where the drone can fly manually or guided. Beyond these boundaries the drone automatically switches its mode to RTL and return to its launch position.

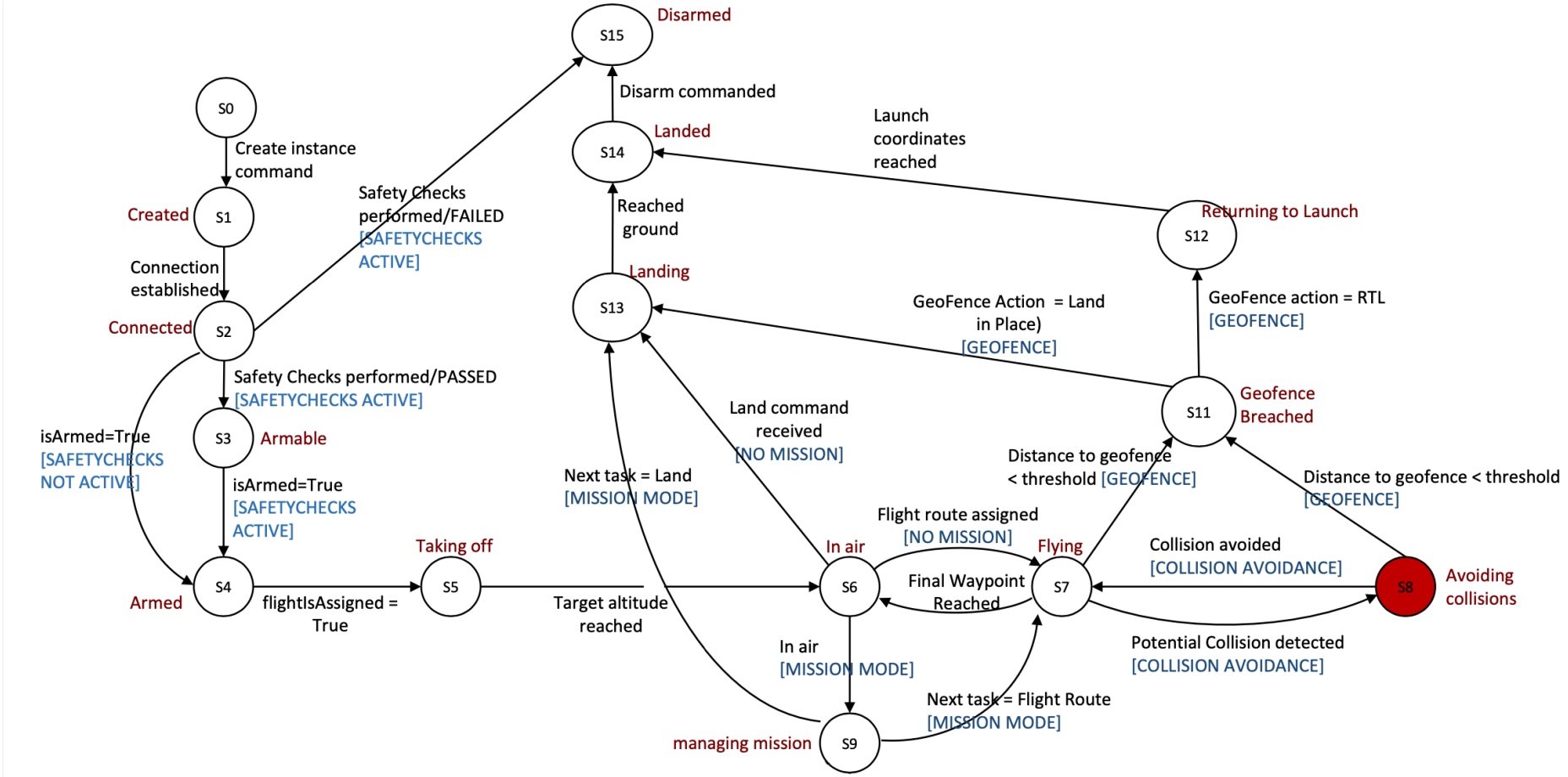

Figure 12 - State diagram of a flight inside Dronology

Explanation

The diagram illustrates an example of a flight using virtual drones. S0 and S1 illustrate the creation of an instance of a drone. For a physical drone, the state S0 could be deleted to keep S01 as ***powered*** and S2 as ***connected***. Once the drone ***connected*** (S2), before arming the drone and launching the flight several checks need to occur to ensure the safety of the drone. These checks are shown in Table 2. The safety check is a feature that is not mandatory it can be disabled, illustrated by the direct transition from S2 to S4. Once the checks are passed, the state passed to ***armable*** (S03) before arming and transit to ***arm*** (S04) once it is armed. In Dronology one can assign a route that has to be followed. Once the route assigned the drone can ***take-off*** (S05) to reach its target altitude. Once the altitude is reached, the drone is ***in air*** S6. One can fly a drone with two modes, mission mode or "normal" mode (no mission mode). The former is explained in 2.2.3.5. The main difference between them is that the latter is not automated. One can fly the same path, however, the "normal" mode will need much more precaution and effort. None of the steps of the flight are controlled, they need to be checked and verified. Example: the takeoff altitude needs to be checked if it is reached, if the waypoint is reached, which waypoint fly next, etc. All those examples are handled automatically by the autopilot if the Mission mode is enabled.

From S6, if the mission mode is active then the state will transit to S9 ***managing mission***. If the last command of the mission is not landing but *flight route*, it means that the flight is ongoing and the drone is ***flying*** (S7). Once a waypoint is reached, the state will transit from flying S7 to S6 in air, then the state transits to S9. From S9, if the waypoint was not the last it will transit to S7 and start again otherwise the waypoint is the last one which implies the landing command being sent. Therefore the state changes from *managing mission* S9 to ***landing*** (S13), then ***land*** (S14) and finally ***disarmed*** (S15). Note that before state S9, considering the mode to be in Mission, the state transit automatically from *in air* S6 to ***managing mission*** S9.

From S6, if the mission mode is not active, the drone will simply fly the assigned route and the state transit to ***flying*** S07. If the waypoint is reached, the last and the next command is land. The drone will transit from in air S6 to ***landing*** (S13), then ***land*** (S14) and finally *disarmed* (S15). If the next command is not land, it means that the flight route is not finished, the drone keeps ***flying*** and the state transits to S7. Note that it is possible that the route is finished but the land command is not given, the drone will simply hover place and be ***in air*** S6.

While ***flying*** S7, it is possible that the drone is getting too close to the geofence and potentially breach it. If the geofence is ***breached***, then the drone enters to state S11. Two actions are possible, depending on the setup of the drone, either a land command is sent and the drone ***is landing*** (S13) where it is. Or a return to launch command is given, then the drone is ***returning to launch*** (S12) before ***landing*** (S13).

If the collision avoidance is active, while flying S7 the drone could face an obstacle, the state transit to S8 avoid the collision and get back to S7. Note that the S8 is in red because the state could be more specific and could lead to a state chart of its own. While the collision is being avoided, it is possible that the geofence is *breached* S11. Which leads to either immediately land or return to launch.

Table 2 - Pre-arm safety checks

Failure type	Message	Description
RC	RC not calibrated	
Barometer	Barometer not healthy	Sign of hardware failure
	Alt disparity	Barometer altitude disagrees with the INS attitude by more than 2 meters.
Compass	Compass not healthy	Sign of hardware failure
	Compass not calibrated	
	Compass offsets too high	
	Check mag field	Sensed magnetic field is 35% higher than expected
	Compasses inconsistent	The internal and external compasses are pointing in different directions (off by > 45degrees).
GPS	GPS glitch	
	Need 3D fix	
	Bad velocity	Vehicle velocity is above 50cm/s
	High GPS HDOP	GPS's HDOP value (a measure of the position accuracy) is above
Inertial Navigation Systems (INS): Accelerometer and Gyro	INS not healthy	
	Accel not healthy	
	Accels inconsistent	the accelerometers are reporting accelerations which are different by at least 1m/s/s.
	Gyros not healthy	Hardware issue
	Gyro calibration failed	gyro calibration failed to capture offsets
	Gyros inconsistent	two gyroscopes are reporting vehicle rotation rates that differ by more than 20deg/sec.
Board Voltage	Check board voltage	the board's internal voltage is below 4.3 Volts or above 5.8 Volts.
Parameter	Ch7&Ch8 opt cannot be same	

	Check FS_THR_VALUE	Radio failsafe pwn value has been set too close to the throttle channels minimum.
	Check ANGLE_MAX	the ANGLE_MAX parameter which controls the vehicle's maximum lean angle has been set below 10 degrees or above 80 degrees.
	ACRO_BAL_ROLL/PITCH	the ACRO_BAL_ROLL parameter is higher than the Stabilize Roll P and/or vice versa.
Battery	Below failsafe	
Airspeed	Failed	If an airspeed sensor is configured, and it is not providing a reading or failed to calibrate, this check will fail.
Logging	Failed to write	
	No SD Card	
Safety switch	Hardware safety switch not pushed	
System	Param storage failed	A check of reading the parameter storage area failed.
	Internal error	An internal error has occurred. Try rebooting.
	KDECAN failed	KDECAN system failure
	UAVCAN failed	UAVCAN system failure
Mission	No mission library present	Mission checking is enabled, but no mission is loaded.
	No rally library present	
	Missing mission item	

2.2 Autopilot

One critical component of an autonomous is the autopilot. It will help users to fly the drone and handle the commands sent from the ground. The ground can be software artefacts (e.g. a small piece of code) or a control system such as MissionPlanner, QGroundControl. Autopilot will make the drone will fly towards the destination without user interaction. The autopilot is a firmware running on the microcontroller of the vehicle responsible for navigation using several sensors like a gyroscope, GPS, accelerometer, barometer, one or more compasses.
It is also responsible for detecting critical situations such as low battery level, obstacles and responding accordingly. Each autopilot has several flight modes that are quite similar. They will determine the autopilot's flight behavior to wind, obstacles and handle the commands sent from the ground. Each flight mode gives a certain level of assistance to the user. They can help stabilize the drone against wind, execute the take-off or landing autonomously, or simply call the drone to his landing position in case of an emergency, etc. There are several autopilots available, however, I have worked with two of the most popular among them, Ardupilot and PX4.

Ardupilot

Ardupilot (APM) is the first autopilot created as an open source, under the BSD license, created in 2009, for unmanned vehicles. It is considered as a pioneer in the domain. Ardupilot began on an Arduino hardware, hence the name "ardu" pilot. The vehicle type is not only limited to quadcopters but also includes plane, submarine, copter and rover.

PX4

PX4 is also an open source firmware like Ardupilot but under BSD it is much younger as it has been released in 2012. PX4 is part of the DroneCode project, funded by the Linux Foundation, which aims to create a suite around UAVs.

There is not much difference between the two firmware except the supported hardware. PX4 has originally been created for a specific microcontroller called PX4. This makes its firmware less compatible with other hardware. Ardupilot, however, supports more hardware. One major key strength of Ardupilot is its stabilization algorithm that is much more advanced than PX4. It results in a better stabilization against the wind. Both of them have strengths and weaknesses. The choice of using one rather than the other one is generally fixed by the manufacturer. Both come with their ground station software.

One of the big differences with Ardupilot is that PX4 is supported by the industry. That is why PX4 is recommended with Intel Aero. PX4 has much more funding because of its license. It is licensed under BSD, which means any modification brought to the software does not need to be shared. Which makes the autopilot mode attractive to a corporation that wasn't to add features or enhance the autopilot.

2.2.1 Flight Controller Unit

The Flight Controller Unit (FCU) is the hardware that will manage every component of the drone like sensors, motors and also the autopilot. The FCU also integrates the basic sensors like a gyroscope, accelerometers and more advance depending on the hardware. It has hubs to host other peripheric. The FCU can be compared at the brain of the drone, something crucial. The Autopilot is a software that is coming as firmware, and is designed to run on a specific FCU. PX4, for instance, has been created first to run FCU called Pixhawk (Figure 13). The project in which PX4 has been designed, also created the hardware. The choice of the FCU depends on factors like the price, the firmware supported, the integrated processor, etc.

Figure 13 - Pixhawk 4 [35]

2.2.2 Ground Station Software

A ground controller station is a software that runs on a ground-based computer also called companion computer. It is responsible for monitoring and communicating with the autopilot by sending commands, as well as providing access to the setup parameters and debugger features. It allows to monitor real-time data received by telemetry (e.g. the position, the battery voltage, speed, the rotors spinning value, etc.). It can be compared to a virtual cockpit.

UAVs can also be controlled by the Ground Station software, which allows sending commands and points to fly by.

There are several control software available but I have used three of them. These QGroundControl Station, Mission Planner and MAVProxy.

They have first been written for a specific Autopilot. QGroundcontrol has been written to support PX4, MAVProxy and MissionPlanner for Ardupilot. The map can also be used to plan missions. It intends to be minimalist and portable while the other two aim to be full-flight controller and mission planning software.

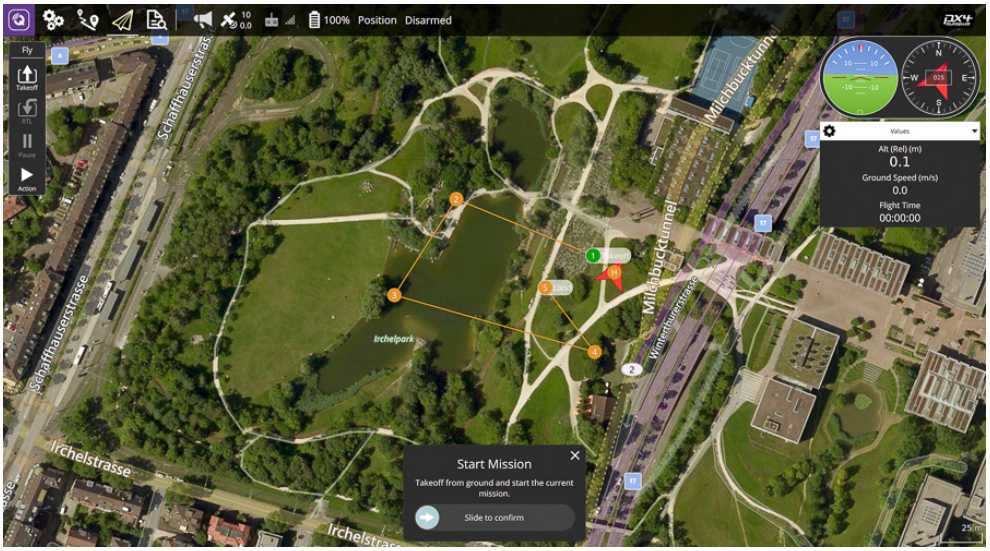

Figure 14 – Illustration of QGroundControl [36]

Initially QGroundControl and MissionPlanner have been designed for a specific autopilot but they are now compatible with both APM and PX4. MAVProxy is only compatible with Ardupilot. Figure 15 is showing Mission Planner with three waypoints and two commands (take-off and return to launch).

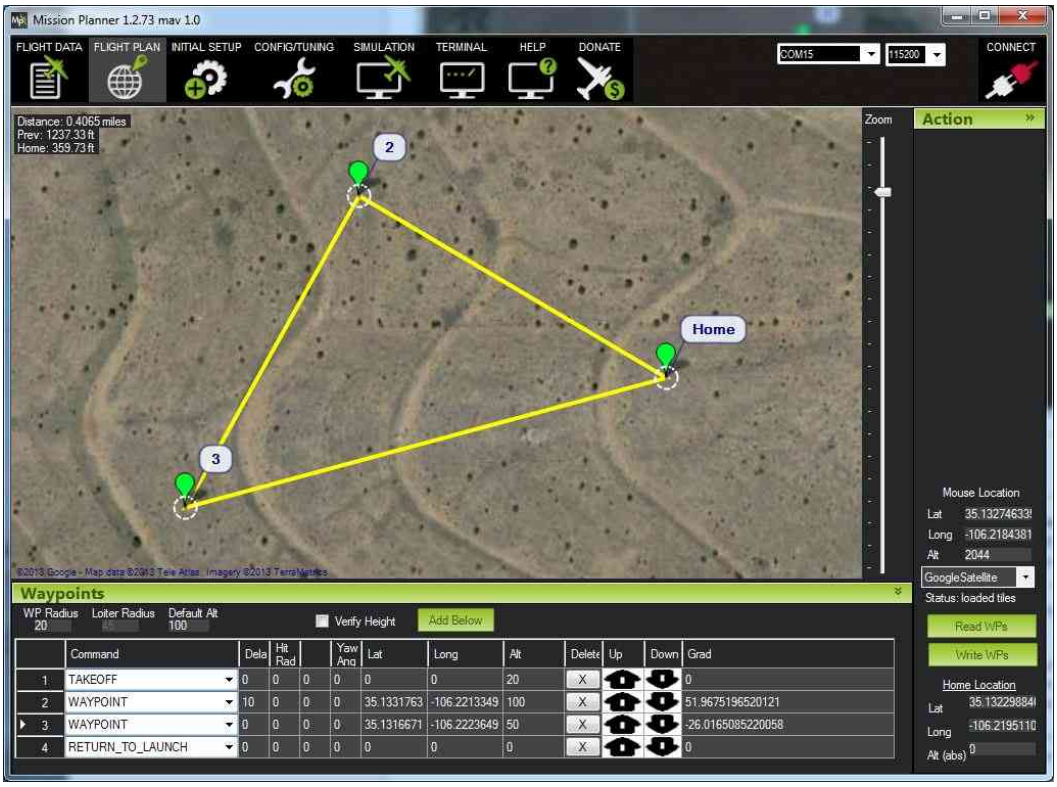

Figure 15 - Illustration of Mission Planner

2.2.3 Flight mode

All the autopilots provide different flight modes. Each of them giving a certain level of assistance from the autopilot to the user. The Autopilot will assist users to fly the drone and define the behavior to user input and responds accordingly depending on the flight mode selected. The flight mode also depends on the type of aircraft used. Neither the number of modes nor the behavior of a flight mode will be the same for a multicopter or a fixed-wing plane. Some of the flights will only be available on multicopter because of the simplicity of the frame.

The next section contains an enumeration of flight modes with their name and their definitions within each autopilot. They are very similar between autopilots. For most of them only the name changes. The flight mode described next does not mention all of the existent but only the most used. Each title represents the mode's name for Ardupilot/PX4. The RC controller had a switch where 3 modes can be configured. Usually AltHold Loiter and RTL are the modes used.

2.2.3.1 Guided/Offboard

Guided/Offboard mode is used for autonomous drone flights. It will allow the autopilot to receive commands from the companion computer. It can be from a ground station software or software piece of software. The instructions will be executed without user interaction

needed. The RC controller is disabled, meaning the roll pitch and yaw cannot be controlled, while the mode is active.

2.2.3.2 Loiter/Hold

Loiter is the mode that stabilizes the drone in the air. When the controller sticks are released the drone will "brake" and hold its altitude and the position using the GPS. The speed is also limited by defaults to 5 ms/s. It is the mode where it is the easiest to fly with manually. It is also the mode used to regain control during a mission in case of emergency.

2.2.3.3 AltHold

Althold is used before switching to Guided mode. It is quite like the Loiter mode, however, the autopilot provides less assistance. The altitude is held by the autopilot but the position needs to be controlled by the pilot. The roll, pitch and yaw are controlled by the user.

2.2.3.4 Return to Launch (RTL)

Return to Launch is used for calling the drone to its take-off location. The autopilot saves the take-off position automatically to come back later if needed.
It is usually used when the drone is too far and almost out of the visual field which makes it difficult and dangerous to be controlled manually.

2.2.3.5 Auto/Mission mode

In Mission mode, the UAV will follow a preprogrammed flight script stored in the autopilot. The script contains commands and waypoints to be followed by the autopilot. One can create a mission both programmatically or using the ground station software. The autopilot will follow the commands one after the other. A mission will start with the take-off that is included by default as the first command, and end with the last command. The last command can either be *land* or *RTL*. However it is not mandatory, if none of the two are specified then the drone will simply hover in place (at the last waypoint reached).
There are two requirements before switching to the mode :

- If the UAV is on the ground, the drone needs to be armed.
- the throttle needs to be all the way down before switching the mode.

A mission starts with the pilot raising the throttle to enable the Mission. If the drone is in the air then the mission immediately starts and the first command is executed.

2.3 Simulator

Several ways are available to make sure that codes written for instance to control UAVs meet its requirement. Such as unitary tests to assess methods consistency, integration testing for module integration, etc. For drone development unit testing is not always useful as methods controlling the UAVs do not compute anything. Instead the software, aiming to control the UAV, creates a behavior that needs to be checked. The safest way to experiment is to test it within the simulator. The advantages of a simulation are being less prone to lead to damages and thus costs. There are several simulators available. I have used two of the most popular ones.

Simulators and autopilots are directly linked: not all autopilots are supported by all simulators and vice versa. Therefore the simulator depends directly on the autopilot used.

I first began to work with PX4. The recommended simulator was called Gazebo. It is also one of the most used which makes it easy to find documentation or support.

2.3.1 Software in the loop

Software in the loop is the term used to describe a modelling technique that provides a generated environment to test executable codes such as algorithms or control systems, etc. The simulations are run to assess the reliability of software towards its specification.

2.3.2 Gazebo

Gazebo is a powerful 3D simulator mainly used for robots. It can simulate outdoor or indoor conditions, allowing users to test algorithms, robot design and behavior, AI systems within a realistic scenario, etc. It has advanced 3D graphics that allow adding models, shadows, lighting effects. It also allows to create and add plugin to enhance the simulator.

Moreover, ROS (see section 2.5) is well integrated with Gazebo. It has a wrapper that allows Gazebo components to directly interact with ROS components. However, Gazebo does not provide the flight dynamic of an aircraft, it is only providing the interface to represent the drone. As well as outdoor and indoor condition such as wind, natural environment with a physical engine, etc.

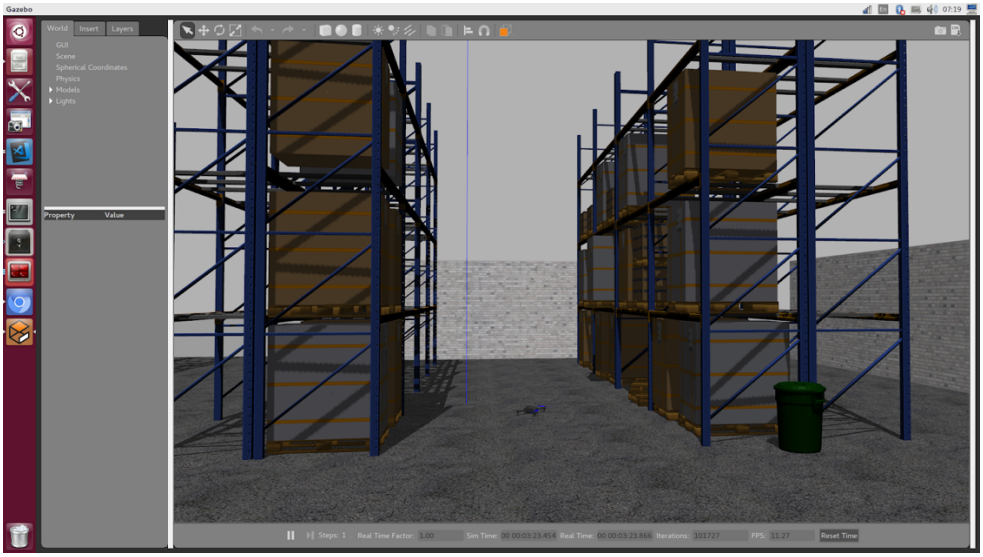

Figure 16 - A flying UAV inside a warehouse. [37].

2.3.3 Ardupilot SITL & MAVProxy

The second simulator used with Ardupilot is called Ardupilot SITL, it is launched by command line with options. In opposite to Gazebo, Ardupilot SITL is a simulator of aircraft. It will create the flight dynamic of an airplane. The flight dynamics are provided by a component called FlightGear [38]. Ardupilot SITL can simulate several types of aircraft, each of them has a different name :

- ArduCopter : for multicopter UAV [39]
- ArduPlane : for winged aircraft
- ArduSub : for underwater vehicle
- ArduRover : for rover

SITL does not provide an advanced 3D environment. But has a simple 2D interface (called MAVProxy) to represent the drone on a map. MAVProxy has the same purpose as a GCS because it allows users to send a command directly to the drone. SITL also provides a set of tools, to help the development of software, such as debuggers, static and dynamic analyzer tools. MAVProxy and the Autopilot communicate with a TCP link on port 5760. It also acts as a bridge between other software (it can be a ground control station, a ROS component) and the autopilot. SITL will use MAVProxy to forward packets to GCS, the communication is done using UDP protocol. An example of SITL (ArduCopter) running with MAVProxy interface is available in Figure 17.

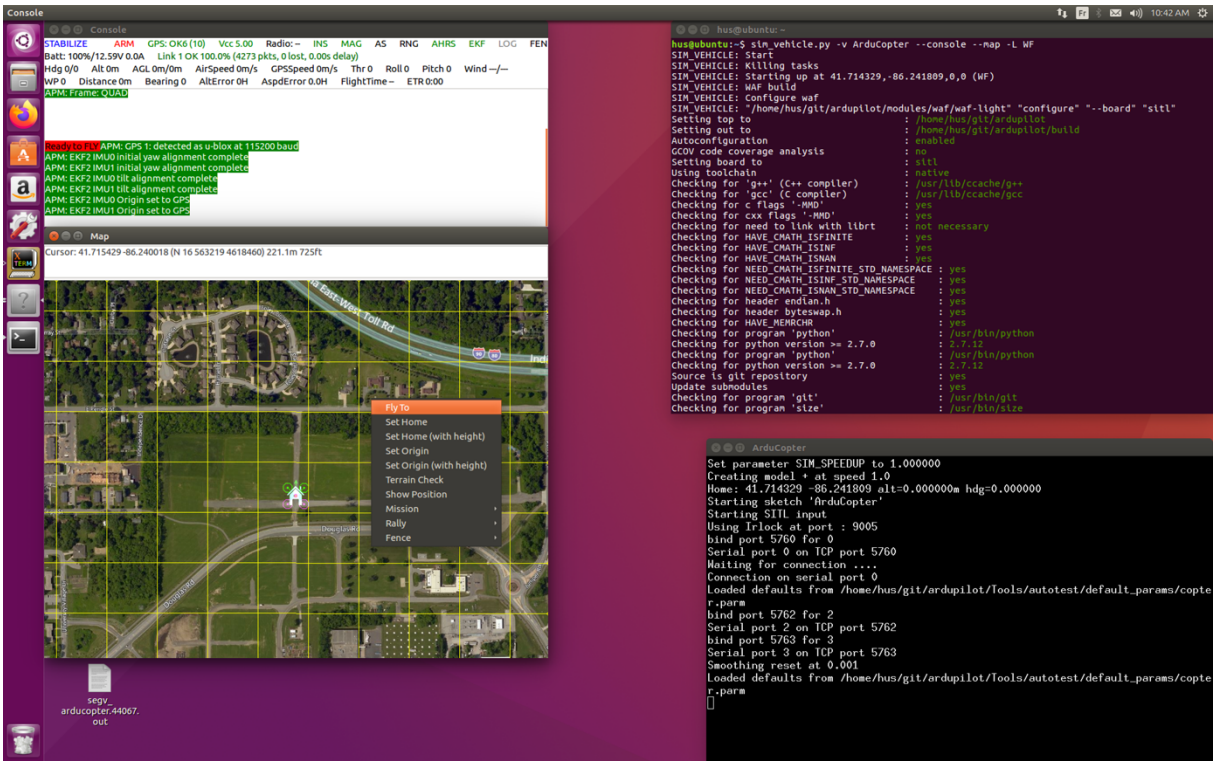

Figure 17 - SITL and MAVProxy running.

While Gazebo can simulate robots and help create environments, SITL will only create the flight dynamic of a UAV. SITL **creates** a simulation of Ardupilot only. While Gazebo can **represent** both autopilot (Ardupilot and PX4).

SITL can be seen as the engine that creates the dynamics of the vehicle and Gazebo as the mechanism responsible for the 3D modeling and the visualization. Both can be combined, Gazebo can be used to create the visualization and SITL to run the autopilot.

2.3.4 Dronekit

Dronekit [40] is a framework that aims to create drone software written in Python. It has a set of development tools. It offers an SDK and API available for Android and Python. The API will allow developers to write applications in Python that will communicate and control the drone. The framework is meant to be used on the companion computer to write software and not on the drone directly. A companion computer is a computer hosting the software that will communicate with the drone. A more detailed explanation is given in section 2.2.2. In this context, Dronology will be hosted on the companion computer. The communication between the companion computer and the drone is established with Telemetry.

2.4 Dronology

2.4.1 Introduction

Dronology [41] is an open source software developed by the Computer Science and Engineering Department of the University of Notre Dame, South Bend, Indiana for Unmanned Aerial System. The Dronology software project started in 2017 under the supervision of Professor Jane Cleland-Huang and Dr. Michael Vierhauser. Prof. Huang is leading the project while Dr. Vierhauser is leading the development as Chief Architect [42]. The software is still ongoing. The idea of developing a software for supervising UAV and studying their behavior comes from the study of safety-critical systems. Safety-critical systems are defined as "those systems whose failure could result in loss of life, significant property damage, or damage to the environment". [43]. Drones are part of those systems along with medical equipment, flight control systems, etc. The lack of data and software environment that allows testing such systems has led to the development of Dronology.

Dronology's goals are "First, to build a safe, deployed, working system in conjunction with real end-users, and second, to create and maintain a research incubator that meets the needs of researchers working in diverse areas of safety-critical systems and cyber-physical systems." [41]. Dronology is a software for controlling, managing, monitoring and coordinating flights. Moreover, it aim to be an incubator that aims to offer a project environment to study different aspects of software development for cyber-physical flying systems such as drones. It integrates a high-fidelity called SITL. Dronology allows researchers to test and verify their solution in a safe environment.

Dronology has already been used for research by the University Of Notre Dame for a search-and-rescue context and a defibrillator delivery solution called DeLive. [44]

The safety aspect of the software is a major concern, as Dronology aims to test safety-critical systems. It needs to be safe. Therefore, research is being conducted on Dronology such as Preliminary Hazard Analysis (PHA), Failure Mode Effects and Criticality Analysis, and Safety Requirements to ensure that the safety requirements of Dronology are met.

2.4.1.1 Supported Research Areas

There are different topic areas that Dronology would support, here are some examples.

- **Software and Systems Requirement :** Dronology creates an environment where data models, system requirements, behavioral views of the system can be studied.
- **Safety Assurance :** most of the CPS are safety critical where systems developed for CPS needs to be verified. The Dronology process has a preliminary hazard analysis, a failure model effect criticality analysis (FMECA), mitigating requirements that address functional concerns and architectural solutions such as fault tolerance, performance, and reliability, and Safety Assurance Case which ensure the safety of Dronology.

- **Runtime monitoring and Adaptation :** many CPS needs adaptation at runtime as respond to changes. Dronology provides runtime monitoring that allows researchers to visualize data collected from sensors and infernally generated events such as flight transitions.

2.4.2 Components

The architecture of Dronology is split in four big components as shown in Figure 18 : the core, the service, the user interface and the Ground Controller Station.

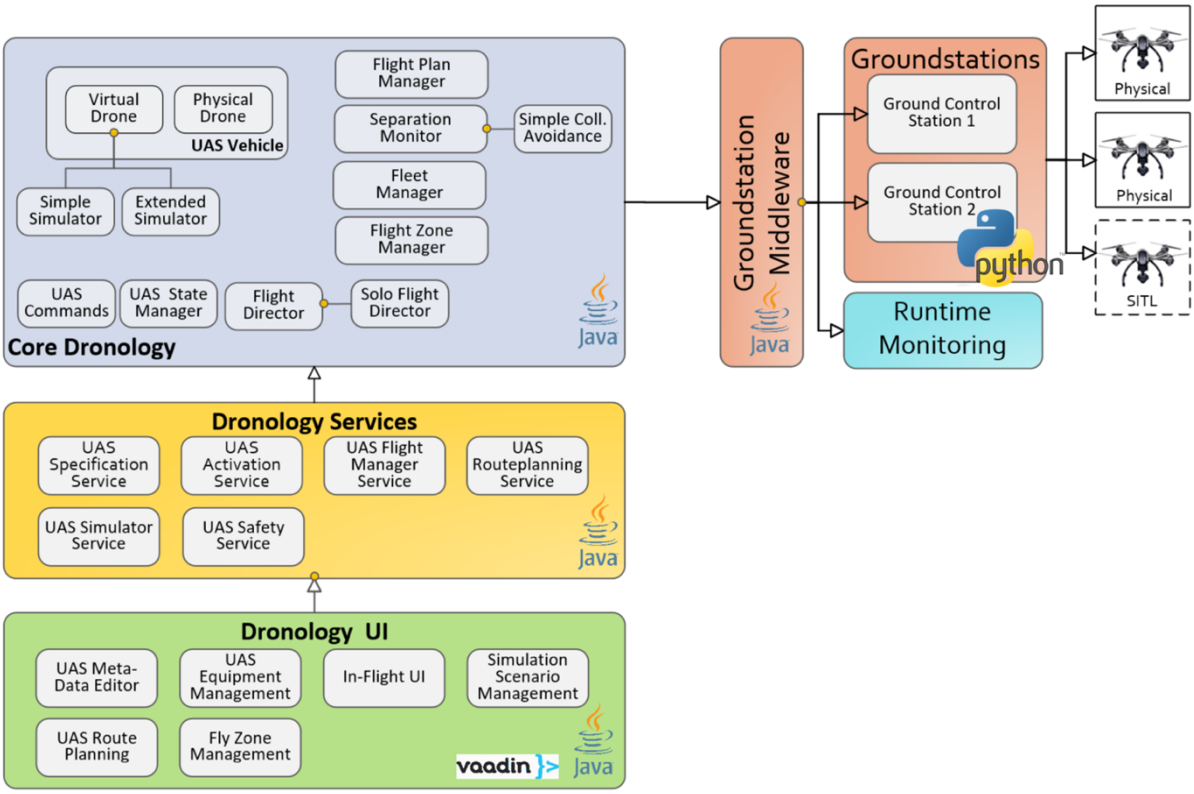

Figure 18 - Dronology architecture

The User Interface

Dronology's frontend is written with Vaadin, an open-source framework for web application development [45]. It allows the implementation of HTML5 user interfaces using Java language. Figure 19 is a screenshot of the main page, where one can see a list of available drones on the left side and their locations on the center. The map is generated using Leaflet, an open-source JavaScript library for interactive maps.

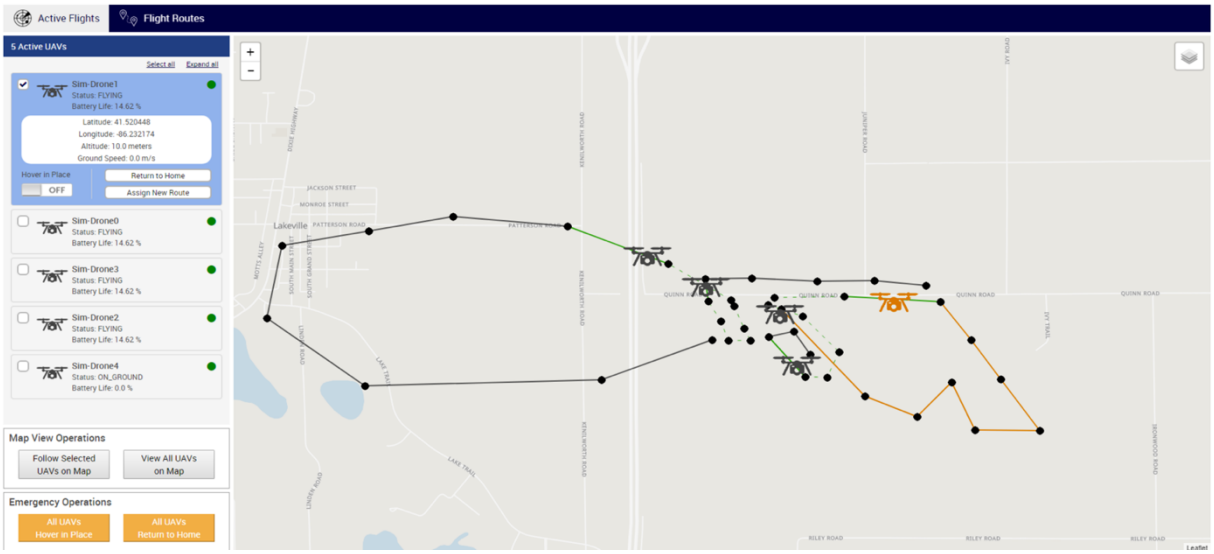

Figure 19 - Dronology UI

The core

The core, or the backend, is written with Java. It is accessible by services. The core is responsible for coordinating flights. It also provides management for vehicles and a simulator (see Simulator section within Dronology) that reproduces drone behavior. The simulation is provided by several simulators, internal to the core and external. The simulators are covered in the simulator section of Dronology's architecture. Dronology implements the primary functionalities that provide everything needed for a safe and quick flight. Those functionalities are : (not exhaustive)

- Establishment of the connection between the computer and the drone using Telemetry (see section 2.1.4).
- Display of the status of the drone, data received by Telemetry (see section 2.1.4).
- Creation of routes that can be assigned to one or several drones.
- Send a return to launch command (see section 2.2.3.4)
- Send a hover command

Groundstation

The GroundStation (GS) is responsible for communicating with the drone directly. Hence, it is responsible for establishing the connection between the backend and the drone as shown in Figure 18. The software is written with Dronekit Python (see section 2.3.4). Its main goal is to create the link between Dronology and the virtual, or physical, UAV. Messages are forwarded from the core to the UAVs through the GS. The core sends messages to the GS that are later transformed to MAVLink messages (see section 2.1.5) and transferred to the vehicle.

GroundStation's role can be compared as a bridge between the software and physical hardware.

The GCS has a small and simple UI that allows one to create a virtual drone or to connect to a physical drone. The virtual drone is created by launching SITL (a simulator of Ardupilot, see section 2.3.3) automatically. Once the virtual drone is created, or the physical drone connected, the core is notified and the drone's information is registered by the core. These information will later be used to communicate with the proper drone. Once the core is notified, the UI is automatically updated to display the registered drone. GCS is mandatory if a physical drone is used since the UI and the core are not directly communicating with the UAVs. The communication with the drones is made by using GCS that is sending MAVLink packets to the UAVs.

Simulator

There are several simulators that allow the creation of virtual drones. Two of them are internal to Dronology written by the University of Notre Dame, a simple and an extended one. They are not much different from each other as they are sharing common code. The extended simulator simply offers more features. The internal simulator is available directly from the UI. One can add drones simply by clicking on a dedicated button.

The external simulator, however, is not integrated in the UI, meaning one cannot create or delete the simulator directly from it. The external simulator is in fact Ardupilot SITL simulator it is therefore not written by the University of Notre Dame.

It is managed by the GroundStation, and responsible for creating an instance of SITL for every virtual drone.

2.5 Robotic Operating System

2.5.1 Introduction

Robotic Operating System known as ROS is a framework providing a set of tools for robotic software development. It is an open source framework, under BSD license, launched in 2017. It aims to be as modular as possible, to allow developers to use parts of ROS that fit the most to the project. Despite the name, ROS is not an OS. However it has several concepts that are shared between OSes. Such as the creation and management of a network of processes, message passing between processes, remote procedure calls, distributed parameters, hardware abstraction, and debug tools, etc. It is available for Linux distribution Ubuntu only and provides libraries to use ROS with Python (rospy) and C++ (roscpp).

2.5.2 Network & ROS Naming convention.

ROS's network is based on a computational graph, a peer-to-peer network of processes that are processing data together. The concepts within the computational Graph are, to name a few: Nodes, Topics, Services, every component that provides data to the graph.
The naming convention used by ROS is organized as a hierarchy tree, applied to all the resources of the network. Each resource is defined within a name space, where other resources can be defined, which introduce a depth. The name spacing mechanism of ROS can be compared to the directory system of Linux. There are four types of names in ROS :

- base
- ~/private/name
- /global/name
- relative/name

The default resolution is done relatively. If the name space of a node1 is /foo and /foo/node1 wants to access to node2, ROS will assume that node2 is under the same name space and will resolve /foo/node2. If the name space is different, then the global name needs to be specified. ROS also provides a mechanism to remap namespace. For instance, to simplify a name or if two instances of a same node is launched they will try to use the same name. Therefore, the second instance will not be launched as the name is already used by the first instance. The names can be the same if the name space if different.

2.5.3 Node

A Node is a process that provides computation. It has a name and needs to register to the Master node. A node can choose to be registered anonymously, therefore the Master will allocate a random generated string to the node. Nodes are meant to be used together to be part of a larger-scale. For instance, a robot can be composed of wheels, camera, sensors and arms. Each of these components can be a node that provides information to each other. The

main benefits of nodes are code reusability. Nodes can be used and reused for building other systems. Fault tolerance, a node that crashes does not necessarily crash the whole system. If one arm of a robot has a defect, the robot can still move with its wheels.
The implementation details of each node are hidden, which helps alternate the implementation of a specific component. The type of wheel can change without bringing changes to other nodes, the way the wheel works does not affect other components of the system. The Node communicates with the Master using the XMLRPC protocol, which is a lightweight stateless HTTP based protocol. The XMLRPC is used by nodes (including the Master node) to communicate with the Master only and negotiate connection with other nodes. It is also used to receive callback from the Master to inform network change. The communication between nodes is done using TCPROS or UDPROS. The former is the most used, a transport layer using TCP/IP socket to transmit data. Every node has a URI composed of the host:port. The negotiation of the transport protocol is made using the XMLRPC protocol.

2.5.4 Master node

The Master node can be described as a ledger, a place where all the names are stored. It does not provide computation but rather plays the role of a DNS within the network and provides a lookup service inside the network. Every node registers their information to the Master node. All the information registered are available to all the nodes of the network. The Master only provides information about nodes. It does not handle the connection between them. The connection is established directly between nodes via Services and Topics. The Master node is launched with the command ***roscore***. The Master node also registers subscribers and publishers of Topics.

2.5.5 Message passing

The ROS typology is based on processes and components that are communicating with each other within a tree graph. Each process takes place inside the network as a node. Nodes can communicate with each other by using ROS messaging interface: topic and services. These interfaces play a middleware role between processes. The messaging system is already in place, allowing to spare time. Process, also node, is the main component written by the developers. It is basically a computation unit that will take action based on received messages. Publish, call or offer service to other components.

Topics

Topics are a way of communication between process, based on the publisher-receiver pattern. A node can subscribe and publish to a topic. Topics are created by the publisher, as shown in Figure 20 at line 7 a topic named chatter is created. The second argument of the constructor

is the data type published. The subscriber is notified when data is published by the publisher. The publisher can publish data to topics for subscribers. The queue size of the topic is defined by the publisher. The writing rate, in Hertz, is also determined by the publisher. A Hertz is a unit that defines the occurrence of an event per second. The queue size and rate values are set at line 7 and 9 of the Figure 20. "Writing" and "Reading" (or consuming) messages are asynchronous, publishers can write messages to topics while messages are being consumed at the same time. If the queue is full, the publisher can publish until space is freed. The subscriber example is illustrated in Figure 21 at line 17. The first parameter is the name of the topic, the second is the data type exchange and the last parameter is a callback function that will be called when a message is published. Topic follows a many-to-many like relationship. A node can subscribe or publish to 0 or more topics. A Topic can also be used to publish and subscribe by several nodes. Publishers and Subscribers are anonymous to each other. One subscriber does not know who the readers are and who the other publishers are if there are any.

```
1  #!/usr/bin/env python
2  # license removed for brevity
3  import rospy
4  from std_msgs.msg import String
5
6  def talker():
7      pub = rospy.Publisher('chatter', String, queue_size=10)
8      rospy.init_node('talker', anonymous=True)
9      rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

Figure 20 - Code snippet of a publisher. [46]

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Figure 21 - Code snippet of a subscriber. [46]

```python
#!/usr/bin/env python

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Figure 22 - Code snippet of a service. [46]

```
int64 a
int64 b
---
int64 sum
```

Figure 23 - Declaration of a service description.

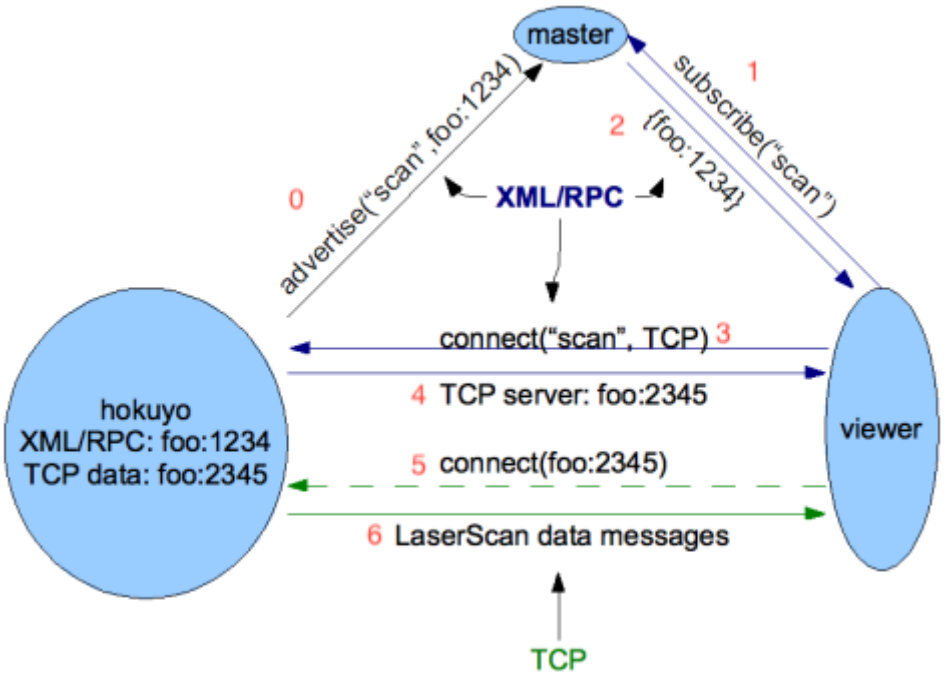

Figure 24 - Illustration of the different steps to exchange messages between two nodes using topic. [47]

Figure 24 is an example of a connection between two nodes. The diagram will sum-up the connection steps described in section Node, Topics and Master Node. Hokuyo is a laser range finder compatible with ROS and publishes data on Scan topic. When a node wants to publish or subscribe to a topic, it needs to notify the Master Node. Step 0, Hokuyo node notifies the Master that it is going to publish on the topic scan and transmits its XMLRPC host name and port. The host name and port will be used by other nodes to negotiate connection. Step 1, the Viewer node asks information about the topic scan to the Master that replies back in step 2 with Hokuyo connection information. Once Viewer obtains the address of Hokuyo, the connection is negotiated in steps 3 to 5. Connection info such as hostname, port and protocol is exchanged during the negotiation. Once the connection is established, the laser starts to publish and the Viewer starts to read (step 6).

Service

Services are another way of interaction between nodes. Services and Topics share the same goal, sending messages. However, Services are synchronous and provide the execution of a procedure. While Topics are only used for communication, Services are used to provide computation or actions. They allow nodes to send requests with values and receive a response accordingly. An example can be found in Figure 22, a service that adds two integer and returns the result. Line 12 is showing the creation of the service. The first parameter is the name of the service, the second is the service type and the third parameter is a callback function to be executed. At line 3 of Figure 22, the object type *AddTwoInts* (the service type), *AddTwoIntsResponse* and *AddTwoIntsRequest* are generated by ROS via a file that describes

the service. The Figure 23 is the description of the service. It contains two parts, the request and the response. The client code is illustrated in Figure 25. The line 10 declares the service, *add_two_ints*, and line 11 calls the service, a request is sent to the service. There is a signature difference between the callback function at line 6 of Figure 22 (takes a *AddTwoIntsRequest* object type) and the call of the service at line 11 of Figure 25 (given two integers). The callback function always takes a request object in which the parameters, given while calling the service, are stored. In the example, at line 11 of Figure 25, two parameters are given X and Y, those will be stored in a request object and be used by the service callback function as illustrated at line 7 of Figure 22. The number of parameters given to the service corresponds to the declaration of the service illustrated in Figure 23. Here X and Y (line 11, Figure 25) corresponds to a and b in Figure 23.

```
1  #!/usr/bin/env python
2
3  import sys
4  import rospy
5  from beginner_tutorials.srv import *
6
7  def add_two_ints_client(x, y):
8      rospy.wait_for_service('add_two_ints')
9      try:
10         add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
11         resp1 = add_two_ints(x, y)
12         return resp1.sum
13     except rospy.ServiceException, e:
14         print "Service call failed: %s"%e
15
16 def usage():
17     return "%s [x y]"%sys.argv[0]
18
19 if __name__ == "__main__":
20     if len(sys.argv) == 3:
21         x = int(sys.argv[1])
22         y = int(sys.argv[2])
23     else:
24         print usage()
25         sys.exit(1)
26     print "Requesting %s+%s"%(x, y)
27     print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Figure 25 - Code snippet of a service call. [46]

Parameter server

Parameter server is a centralized database where values are stored by nodes to be used at runtime. It uses a dictionary of dictionaries where each dictionary corresponds to a namespace. The parameter server is not meant to be high-performance, it will be more used to store static values like configuration parameters. These values are shared between all the nodes inside the network. The parameter server is part of the Master.

2.5.6 Tools

First, one of the main strengths of ROS is its tools set that helps debugging and visualization. ROS has lots of tools that can be displayed as little plugins.

- **Rqt** is the software that allows users to build their own interface. Plugins can be combined with the necessary plugin to build a window.
- **Rqt_graph** is a graphical interface plugin that displays the communication flows that exist between ROS's component within a graph. An example can be found in Figure 31. The oval shape objects are nodes, rectangles are topics. All the topics sharing a same name space are gathered together. An outgoing arrow from a node to a topic means that the node is publishing to the topic. An incoming arrow means subscribing to the topic. *Turtlesim* and *teleop_turle* are two nodes provided by ROS for tutorial purpose. The former is simulating a 2D turtle (Figure 26) and the second is a node that listens the keyboard input and publishing the values to control the turtle. We can see that *teleop_turtle* is publishing velocity commands on topic called /turtle1/*cmd_vel* that is subscribed by *turtlesim*. *Turtlesim* is publishing on topic pose, a topic that controls the movement of the turtle.
- **Rqt_console** which allows users to display the messages published to *rosout* (display log messages).
- **Rqt_topic**, **Rqt_publisher**, both of them are using to interact with topics. The former monitors and inspects the topic that is used in the network. The latter is used to publish to the topic thus simplifying the interactions with them.
- **Rviz** is an advanced 3D visualization tool to represent robots and sensor data types. Rviz allows the user to visualize the data type, coming from sensors, like laser scans, camera images, etc. An example of RVIZ is illustrated in Figure 27, where a robot opens and goes through a door. The left screen is a RVIZ window showing a representation of the robot and the data coming from the sensors.
- **Catkin** to manage package, build and generate codes. It can be used to create services, topics, etc.

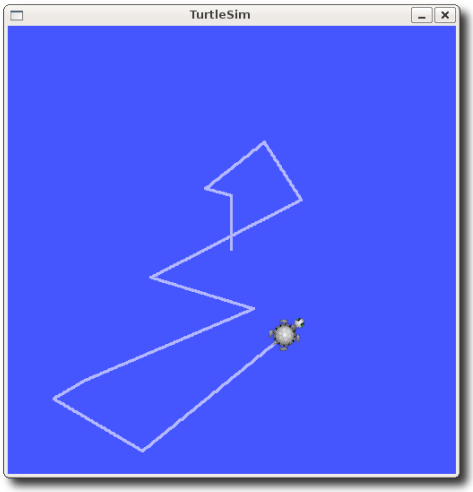

Figure 26 – Turtlesim.

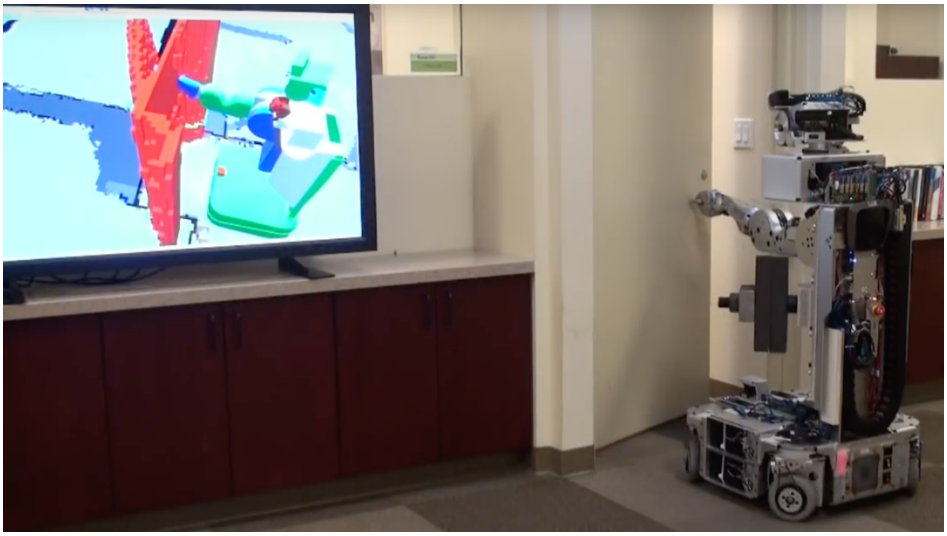

Figure 27 - Rviz in action. Source: YouTube, Rviz official presentation video

Second important point is that every component of the network is accessible via command line.

ROS provides a command line tool to interact with Topics, Services and Node. For instance, list all the services, topics, gather information, retrieve data type, publish data on to topics, listen to data published, etc.

- **Rosservice** to interact with services. The complete list of commands is displayed in Figure 28.
    - **Example**: rosservice call /foo/srv1 true 10, call a service called srv1 with a two parameters.
- **Rosrun** to run Node.
    - **Example**: rosrun /foo/node1, run the node1

- **Rosnode** to display information about nodes. The complete list of commands are displayed in Figure 30.
    - **Example**: rosnode nfo /topic1
- **Rostopic** to interact with topics. The complete list of commands are displayed in Figure 29.
    - **Example**: rostopic echo /foo/topic1, will show all the messages publish to topic1.

```
rosservice call call the service with the provided args
rosservice find find services by service type
rosservice info print information about service
rosservice list list active services
rosservice type print service type
rosservice uri  print service ROSRPC uri
```

Figure 28 - rosservice [48]

```
rostopic bw     display bandwidth used by topic
rostopic delay display delay for topic which has header
rostopic echo   print messages to screen
rostopic find   find topics by type
rostopic hz     display publishing rate of topic
rostopic info   print information about active topic
rostopic list   print information about active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

Figure 29 – rostopic [49]

```
rosnode info    print information about node
rosnode kill    kill a running node
rosnode list    list active nodes
rosnode machine list nodes running on a particular machine or list machines
rosnode ping    test connectivity to node
rosnode cleanup purge registration information of unreachable nodes
```

Figure 30 - rosnode [50]

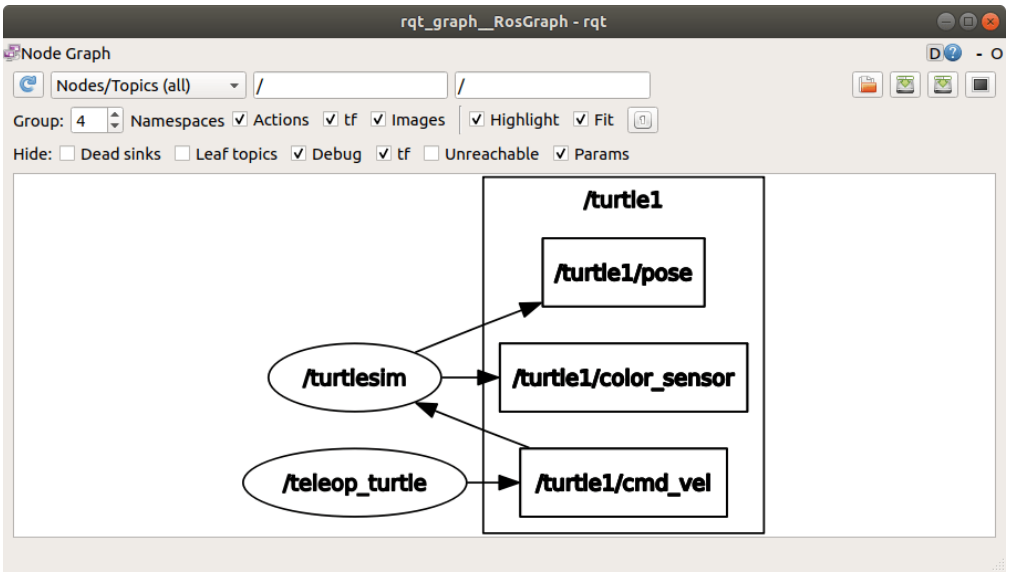

Figure 31 - Rqt_graph of two nodes communicating with each other.

2.5.7 Mavros

MAVROS is a communication node used for sending MAVLink messages to components like ground control station. MAVROS plays a major role in the development as it is offering the necessary notions to interact with the drone, by transmitting the MAVLink commands to the autopilot. As mentioned, MAVROS is a communication node, it can be compared to a bridge between the autopilot and the controller node. The controller node will communicate with MAVROS through topics and services, while the MAVROS transmits the right information to the Autopilot. To achieve this, MAVROS is publishing information such as positions, states of the autopilot to Topics so that subscribers can access. And MAVROS subscribes to Topics in order to provide nodes to send commands to the autopilot.

Figure 32 is a list of the topics, not complete, available in ROS provided by MAVROS. Topics are grouped, by their namespace.

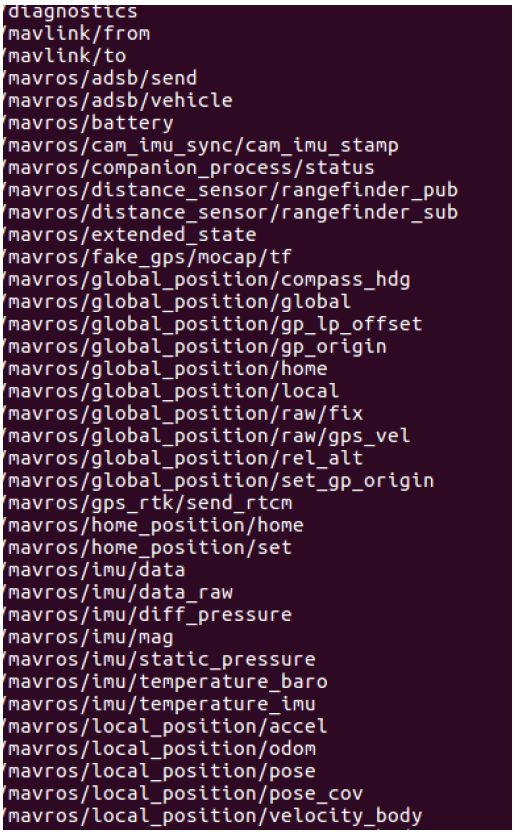

Figure 32 - MAVROS topics

2.5.7.1 Coordinate frame & system

Autopilots biggest benefit is to give commands without worrying much on how it will be executed. Therefore one can provide coordinates and the autopilot will handle the way to reach the destination. There are two ways of expressing the coordinate system used to describe a destination, in ROS, local and global. Topics that will use global positioning system contains "*global_position*" in their name space. And "*local_position*" to use local coordinate system.

The **coordinate frame** is the set of reference lines attached to a body of an object used to describe the position of a point relative to the object. The coordinate frame is a fixed point at a known location. For instance, the position of a point on the surface of Earth is described by a latitude measured from the Equator and the longitude measured from the prime meridian (Greenwich). There are few **coordinate frames** available for aircrafts, here are the four common frames : Each of the frames are represented in Figure 33

- Earth-centered earth-fixed frame : is the coordinate frame used by the GPS. Its origin is the center of earth. The three axes, X Y Z, start from the center of earth. The X axis is pointing to the intersection between the meridian of Greenwich and Equator. The Y

axis pointing along the equator, perpendicular to X and Z-axis. And the Z axis points toward the north pole.

- Local-vertical Local-horizontal frame (LVLH) : the coordinates are interpreted according to the vehicle's position. It is used to describe local trajectory. The origin will represent the center of gravity of the vehicle and all the axis points toward the center of Earth. A local coordinate is expressed by three coordinates, X Y Z, that correspond to the three axes that go through the body of the drone. The X axis points toward the north. The Y axis points east and the Z axis will point toward the center of the earth. Each of these axes is tangent to geographic coordinates. The X axis is tangent to meridians, the Y axis is tangent to parallels and the Z axis is in the direction of the axis that goes through to the center of the globe from the north. For that reason, the LVLH frame is also called the local tangent lane coordinate.
- Body fixed frame : The body frame will also take the center of gravity of the vehicle as its origin. As described in section 2.1.3. The X axis points out of the nose, the Y axis points toward the right wing and the Z axis towards down. They are also known as yaw, roll and pitch.
- The IMU frame : is centered at the location of the Inertia Measurement Unit (IMU). The IMU frame uses three gyroscopes and three accelerometers to determine the orientation of an aircraft. Each gyroscope and accelerometer is aligned with each axis.

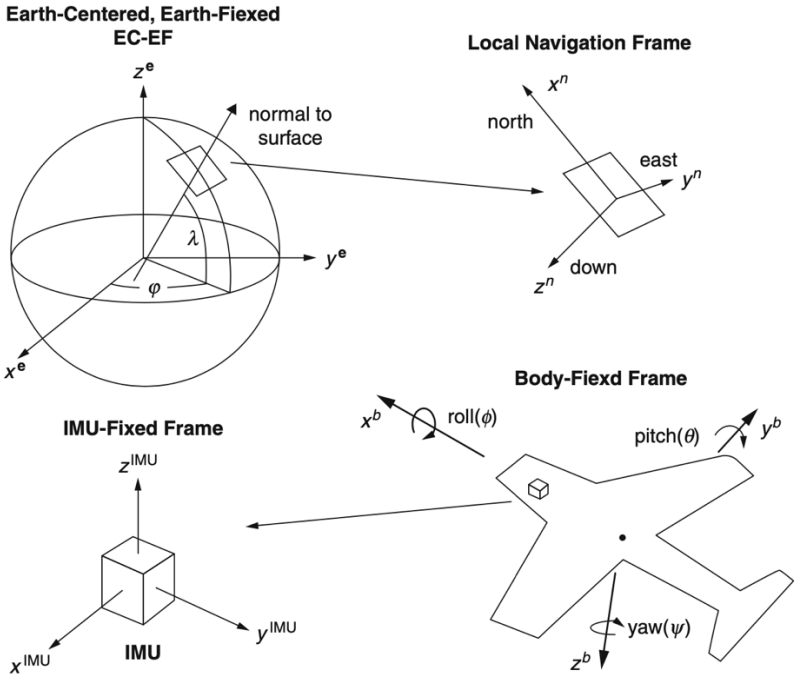

Figure 33 - Representation of each frame of reference: ECEF, LVLH, IMU-fixed and Body-fixed. Source: Handbook of Unmanned Aerial Vehicle. [4]

The **coordinate system** is a set of reference lines used to identify and determine the position in space of an object. For instance, the Cartesian system is a coordinate system that describes the position of a point by its distance along the X axis and its distance along the Y axis, from a reference point (0,0). The position and the orientation of an object is called **Pose**.

2.5.7.1.1 Local coordinate system

"Local" based topics accept only local coordinates, that are local to the vehicle.
A local coordinate is expressed by a *geometry_msgs/Pose* type of message provided by ROS, illustrated in Figure 35. The Pose data type is composed of Point, which represents the position of a point the drone will go toward. And Quaternion coordinates that represent the orientation and rotation of the vehicle.

```
hus@ubuntu:~$ rosmsg show mavros_msgs/GlobalPositionTarget
uint8 FRAME_GLOBAL_INT=5
uint8 FRAME_GLOBAL_REL_ALT=6
uint8 FRAME_GLOBAL_TERRAIN_ALT=11
uint16 IGNORE_LATITUDE=1
uint16 IGNORE_LONGITUDE=2
uint16 IGNORE_ALTITUDE=4
uint16 IGNORE_VX=8
uint16 IGNORE_VY=16
uint16 IGNORE_VZ=32
uint16 IGNORE_AFX=64
uint16 IGNORE_AFY=128
uint16 IGNORE_AFZ=256
uint16 FORCE=512
uint16 IGNORE_YAW=1024
uint16 IGNORE_YAW_RATE=2048
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint8 coordinate_frame
uint16 type_mask
float64 latitude
float64 longitude
float32 altitude
geometry_msgs/Vector3 velocity
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 acceleration_or_force
  float64 x
  float64 y
  float64 z
float32 yaw
float32 yaw_rate
```

Figure 34 - GlobalPositionTarget message uses as GPS coordinates.

```
hus@ubuntu:~$ rosmsg info geometry_msgs/PoseStamped
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

Figure 35 - PoseStamped type message use as local coordinate

2.5.7.1.2 Global coordinate system.

Global position corresponds to the standard positioning system used in most of our devices called GPS, determined by a latitude, longitude and altitude. A global position is expressed by the message type called *sensors_msgs/NavSatFix* which is composed of Latitude, Longitude and Altitude. However, destination targets are provided by *GlobalPositionTarget*, in Figure 34, that contains the GPS coordinate. The altitude can also be interpreted in different ways, it is specified by the *coordinate_frame* attribute. It describes the positioning system that has to be taken into account.
It has three values, for the global coordinate system, the first three lines in Figure 34 :

- FRAME_GLOBAL_INT
- FRAME_GLOBAL_RELATIVE_ALT
- FRAME_GLOBAL_TERRAIN_ALT

All of them are using a coordinate system called the Global Geodetic System 1984 (WGS 84), that is used by the GPS. The main difference between them is the interpretation of the altitude.
The FRAME_GLOBAL_RELATIVE_ALT altitude is relative to the home position of the object. It means that the altitude of an object at home position is equal to 0.
FRAME_GLOBAL_INT is using the Mean Sea Level (MSL) as reference for the altitude.
And finally, the FRAME_GLOBAL_TERRAIN_ALT is calculating the altitude from the ground level (AGL). It is still relative to the vehicle, however, the altitude will be adjusted to the elevation of the ground. This is a particular frame only usable with a Ground control software and a particular mode called Terrain. In this mode, the autopilot will adjust the target altitude according to the elevation of the ground. For instance, if a target point is on a hill the autopilot will adjust the height of the vehicle. This adjustment is made using with the data provided by the ground station software. Otherwise, without adjustment, the drone may crash.
Target point is transmitted through Topics, in Figure 36, where MAVROS subscribes to get published data so it can transmit it to the Autopilot.
The type_mask attribute is used to specify which attribute is ignored. The value is the sum of all the attributes ignored. For instance, if the velocity, acceleration, force and yaw are not used than the value will equal to 4088.

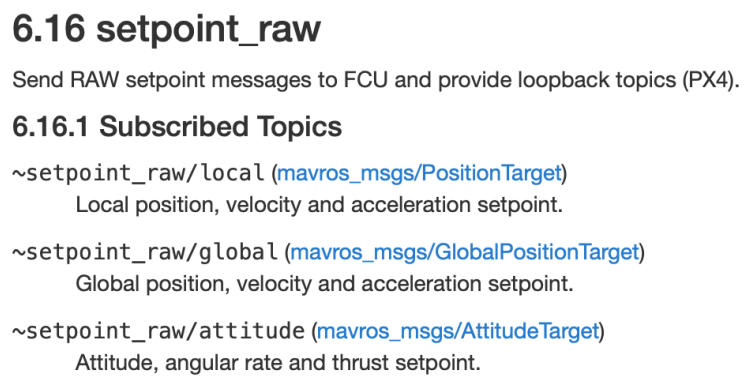

Figure 36 - setpoint_position topics. [47]

2.5.7.2 Modes

The mode in which the autopilot is operating can be switched using services, illustrated in Figure 37. It takes an integer in argument which represents the mode and returns a Boolean

that return True if it is a success, False otherwise. All the base modes, in Figure 39, are modes provided by ROS directly and custom modes by the autopilot. When the custom modes, in Figure 38, are used the base mode value needs to be equal to 0. The integer corresponding can be found on the website of each autopilot.

```
hus@ubuntu:~$ rosservice info  /mavros/set_mode
Node: /mavros
URI: rosrpc://ubuntu:47282
Type: mavros_msgs/SetMode
Args: base_mode custom_mode
```

Figure 37 - /Mavros/set_mode service

The service takes two arguments, the first one is the base mode generally. It is always on PREFLIGHT or 0 and the second value is specific to the mode wanted.
Once the mode is set to the proper mode, commands can be sent. Otherwise, any commands will be rejected with an error message notifying that the mode does not allow the command.

```
uint8 MAV_MODE_PREFLIGHT=0
uint8 MAV_MODE_STABILIZE_DISARMED=80
uint8 MAV_MODE_STABILIZE_ARMED=208
uint8 MAV_MODE_MANUAL_DISARMED=64
uint8 MAV_MODE_MANUAL_ARMED=192
uint8 MAV_MODE_GUIDED_DISARMED=88
uint8 MAV_MODE_GUIDED_ARMED=216
uint8 MAV_MODE_AUTO_DISARMED=92
uint8 MAV_MODE_AUTO_ARMED=220
uint8 MAV_MODE_TEST_DISARMED=66
uint8 MAV_MODE_TEST_ARMED=194
uint8 base_mode
string custom_mode
```

Figure 39 – Base Mode. [51]

0	STABILIZE
1	ACRO
2	ALT_HOLD
3	AUTO
4	GUIDED
5	LOITER
6	RTL
7	CIRCLE
8	POSITION
9	LAND
10	OF_LOITER
11	DRIFT
13	SPORT
14	FLIP
15	AUTOTUNE
16	POSHOLD
17	BRAKE
18	THROW
19	AVOID_ADSB
20	GUIDED_NOGPS

Figure 38 - Custom mode provided by Ardupilot. [58]

Chapter 3 Contribution

3.1 Introduction

The main objective of this work was to explore ROS (see Section 2.5), use it for drone flights and observe the behavior. To achieve the goal, several flights of drones have been performed within different scenarios using one to three drones.
Before the start of the development step, first I had to start a learning curve mainly about the drone universe in general then with ROS.
I had few knowledge about some electronic components like Phidget or Arduino thanks to some projects of the University of Namur, but nothing about drones.
So the first step towards getting familiar with the drone universe was to participate in outdoor flights with Prof. Cleland-Huang's team to test and prepare the material for the drone class. It helped me getting directly in contact with drones and the materials that surround them. But also with Dronology, which had been used during these outdoor tests. And therefore allowed me to see and interact with Dronology.
The second major step was to start getting acquainted with ROS by gathering materials such as official documentation, tutorials, books, etc.
And finally, make the hardware usable with ROS by installing the framework on drones and properly configure them. The major part of my work is focusing on writing pieces of software, or nodes (see section 2.5.7.2), that send commands to drones.
The outdoor tests described in the next sections are always done with the presence of Prof Cleland-Huang, Dr. Vierhauser, referred as we.

3.2 Hardware

The hardware that I used for my tests was new, therefore had to be configured first. What is going to follow in this section is the issues that I have faced during the setup of the hardware. These issues are listed and explained, to draw attention on the fact that hardware can sometimes be difficult to work with. Especially when the hardware is new and do not have a lot of support for it. In such cases, the best solution is to test and experiment.

3.2.1 Drone

The drone that I have used is manufactured by Intel and is called Intel Aero Ready to Fly (Figure 40). It was first presented in 2016 during the Intel Developer Conference in Los Angeles [52]. The Aero was presented for developers and targeted the development of autonomous drones. It comes with several cameras, and integrates a module called Intel Real Sense R200. The module offers depth camera sensing and 3D imaging.

Intel offers an installation guide to properly configure the drone to fly.
Unfortunately, few issues occurred during the installation.
The first issue concerns the cameras. Which I have never been able to make (all) them work.
There are in total five cameras :

- three of them were included in the Intel RealSense R200 module
- Two of them are independent cameras
    - One on the bottom side, a black and white camera
    - The second on the front, a high-definition camera.

The last two never worked. After few researches on forums and websites, I found out that I was not the only one with this issue. And it has not been fixed yet.
During the setup, one of the steps is to flash the autopilot to the flight controller. During the installation, the process was sometimes odd, by being caught in an endless loop. This is completely random and is mentioned in the installation guide, in section Flashing Controller [53]. The solution suggested was to wait 30 seconds or cancel the process and start over. This step could either last during a short period of few seconds or a long period of few minutes.
Once the setup process completed, we decided to test the drones (three). We have tested the drone with both PX4 and Ardupilot. The former is the default recommended by Intel. However we have encountered a few issues.
The first one concerns the stability during flight. The drone was not steady enough during the test. Instead, it was constantly drifting, which makes it unsafe to fly. This behavior can occur with two conditions, when the drones are not properly calibrated or due to wind. On that day, the wind speed was slow enough to fly drones. The weather is always checked before proceeding to tests.
After a few calibration attempts, due to calibration failures, the drone seemed to be stable enough. However not as stable as Ardupilot. The calibration process was also not always stable as it sometimes failed for any particular reason. The second argument to switch to Ardupilot is the connection problem that occurred with Dronekit. Prior the first flight, a script had been executed to check different safety parameters. However, Dronekit could not get any connection to the drone and got a timeout error every time.
These two issues made us switch to the alternative, Ardupilot. Ardupilot is used on every other drone that the faculty has. Therefore switching to Ardupilot was not a difficult choice to make.
Autonomous drone programming is a new field that is starting to expand in consequence. It does not have a lot of documentation as it could be with software development. It takes a significant time to solve a problem because the solution is coming from experimenting with different setups without knowing if it is going to work.

Figure 40 - Intel Aero Ready-to-fly

3.3 Software

This section is the principal part of my work. It concerns my work with Robotic Operating System used for autonomous drone flight development. The two goals of the proof-of-concept to implement the adequate functions to make one or more drone flights and exchange messages during flight.

The functions have been implemented by using ROS and the MAVLink library called MAVROS. It allows sending MAVLink messages to the drone. These functions are tested within the simulator and few of them on outdoor tests. The code is composed of 1336 lines, with 794 of them being source code line.

3.3.1 Architecture & Component

There are few components used to develop the proof of concept. The first component is ROS, providing the environment of execution with topics, services, etc. The software is written in python, using the python client library for ROS called *Rospy*. It allows interfacing with ROS topics, services and other ROS components. ROS cannot send MAVLink messages natively to interact with the autopilot. Instead a library called MAVROS needs to be used. It is an adaptation of MAVLink protocol for ROS. The UAV is either a virtual drone, generated by Ardupilot SITL, or a real drone.

The integration between the virtual drone, the UI controlling the drone and ROS is represented in a high-level diagram in Figure 41. The arrow between two components is the type of data exchanged.

ROS is composed, primarily, with topics and services that will be solicited by the controller node. The controller node is the node responsible for directly controlling the flying UAV by sending commands like, mode switches, coordinate to fly, etc. Examples of topics and services can be found in the next section 2.5.7. The node will communicate with the Autopilot using topics and services provided by MAVROS. The data transmitted by the controller node to MAVROS will be transformed to MAVLink messages and sent to the MAVProxy. When MAVProxy is launched, it has an IP where software like Ground control software to communicate with the autopilot. The communication between MAVROS and MAVProxy is done via an UDP link using the port 14550 as mentioned in section 2.3.3. MAVProxy will forward the incoming packet to the Autopilot using a TCP connection. The port is set by default but can be changed. Each SITL instance will simulate one virtual drone and each MAVROS node will be used to communicate with one drone. A MAVROS node cannot be used to communicate with several drones.

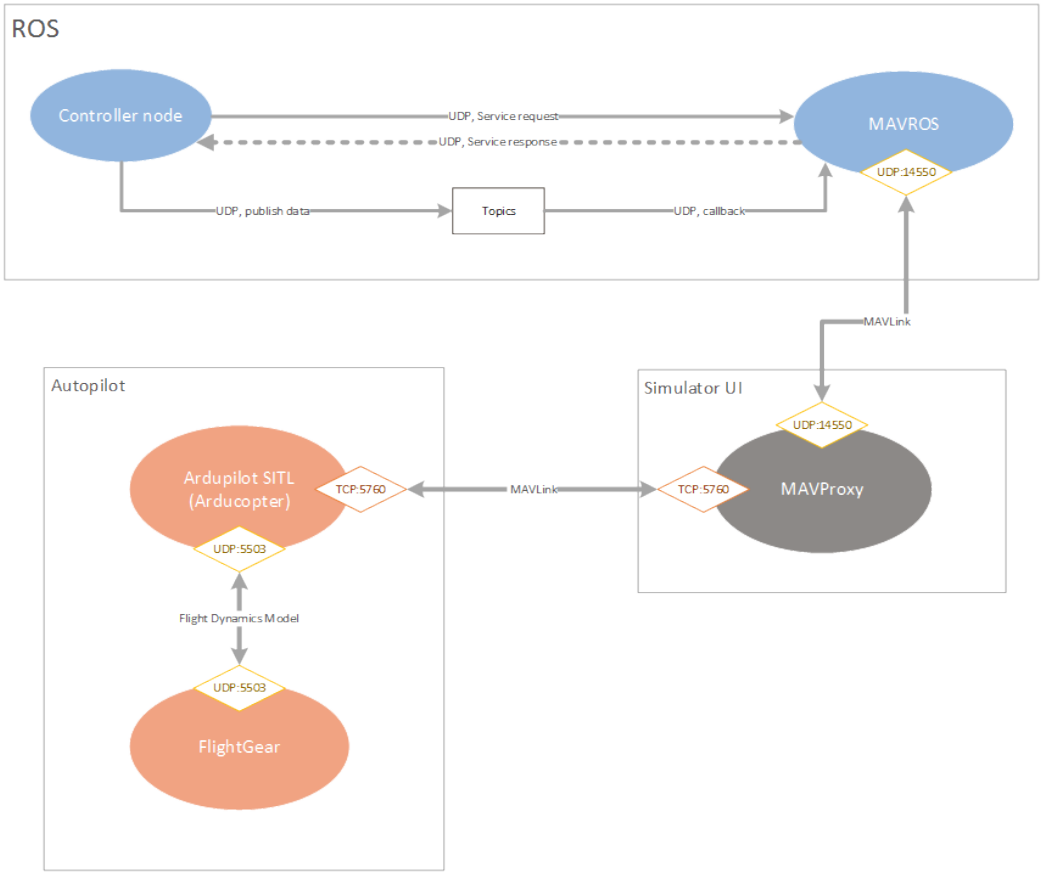

Figure 41 - ROS, MAVProxy, and SITL integration.

3.4 Implementation

3.4.1 Introduction

As mentioned in section 3.3.1, the implementation has been done using Python and an adaptation of ROS for Python, called Rospy. I have first started to implement the code using PX4 as Autopilot before switching to Ardupilot because of issues stated in section 3.2.1.

3.4.2 Code

The code is organized as follows:

- A domain package that contains :
    - A Vehicle class that contains all the topics and services necessary to make the drone fly.
    - A Mission class that creates a fake mission with hardcoded coordinates.
    - A RouteAssignment class that is used to assign route for specific UAV.
- A Util package that contains :
    - A Util class that contains:
        - A method to determine the remaining distance between the actual position of the drone and its target.
        - A method that load a JSON file.
    - GAPesultParser that parse a JSON file coming from a Genetic Algorithm, that contains routes.
    - A flight_plotter class that plots coordinate of a flight.
    - Ned_util that contains methods relative to velocity vector.
- Uav1 class that is used to test implemented function in Vehicle class. The actual state of the class allows launching Mission or normal flight using a single drone.
- FlyMultipleDrone that uses Vehicle class to launch a flight with three drones.
- MasterSlave that launches two drones, where one is following the other one.
- Seams is a class that has been written to test the result of a Genetic Algorithm.
- TestThread class that is used to test the integration of Threads.
- TestNed class to experiment with velocity vectors.
- FlyCircle where the objective was to fly a circle using velocity vectors.
- EvalFieldTest that contains random command to test the behavior of ROS.

The last two classes are experimental and not functional.

3.4.2.1 Vehicle class

All the services and topics needed to communicate with the autopilot are declared in the Vehicle class. It also implements function that uses topics and services in order to send commands to the UAV to make it fly.

The class has been written incrementally starting with little interaction, such as mode changes, arming the UAV, etc, before implementing the actual methods.

The ***init*** function, Figure 44, declares and initializes the attribute of the class. Most of the attributes are composed of services, topics to interact with the drone and attributes.

Services used are :

- /Mavros/set_mode for changing modes
- /Mavros/cmd/arming to arm the rotors
- /Mavros/cmd/take_off for take-off
- /Mavros/cmd/land, for landing.
- /Mavros/mission/push, to add waypoint to the mission.

Topics used as publishers :

- /Mavros/setpoint_raw/global, to give waypoint

Topics used as subscribers :

- /Mavros/global_position/global to get the current position of the UAV.
- /Mavros/state to get the state of the UAV.

Other topics and service not mentioned in the list but present in the snippet are experimental or not used.

These services and topics are called within methods that process data and handle responses. An example is illustrated in Figure 43. It is the simple method that arms the UAV. The method takes two parameters, the first one is the class itself, the second one is the Boolean that is going to be used to either arm or disarm the vehicle. The method is sending a request to the service responsible for arming the drone, "/mavros/arming". The service type is *CommandBool,* Figure 42*,* the request contains one Boolean attribute Value, and the response contains the result composed of a Boolean and an integer.

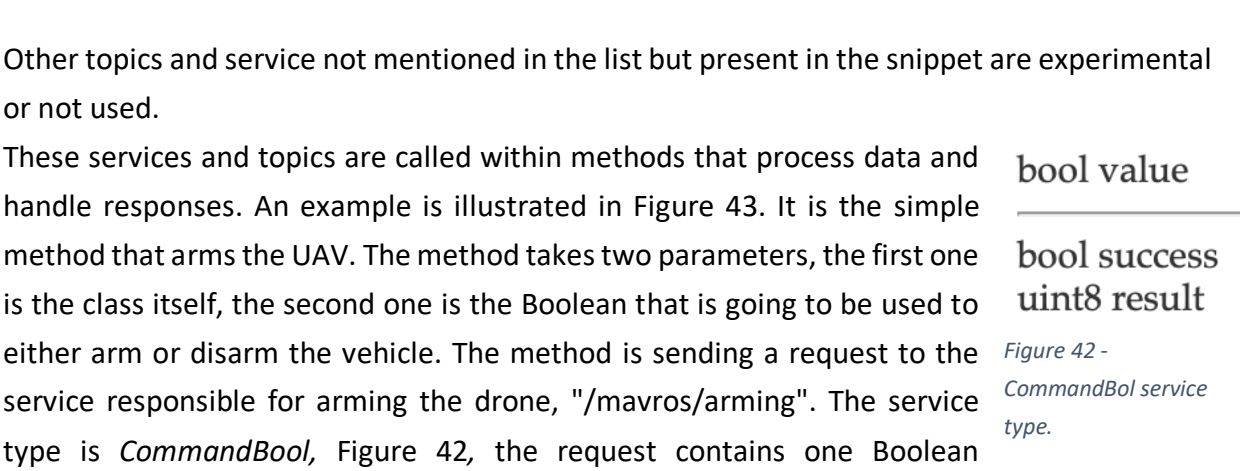

Figure 42 - CommandBol service type.

```
73      def arm(self, arg):
74          rospy.wait_for_service(self.name + '/mavros/cmd/arming',15)
75          try:
76              resp = self.setArm_srv(arg)
77          except rospy.ServiceException as exc:
78              rospy.loginfo("Arming did not process request: " + str(exc))
79          else:
80              rospy.loginfo("Arming : "  + str(resp.success))
```

Figure 43 – arm function that arms the vehicle.

```
16      #Declare all the topics needed
17      def __init__(self, name):
18          self.name = '/' + name
19          self.setMode_srv = rospy.ServiceProxy(name + '/mavros/set_mode',SetMode)
20          self.setArm_srv = rospy.ServiceProxy(name + '/mavros/cmd/arming', CommandBool)
21          self.takeOff_srv = rospy.ServiceProxy(name + '/mavros/cmd/takeoff', CommandTOL)
22          self.land_srv = rospy.ServiceProxy(name + '/mavros/cmd/land', CommandTOL)
23          self.missionPush_srv = rospy.ServiceProxy(name+'/mavros/mission/push', WaypointPush)
24          self.useVelocity_srv = rospy.ServiceProxy(name + '/mavros/setpoint_velocity/mav_frame',SetMavFrame)
25
26          #I've tested the 3 following topics, only one worked properly.
27          self.setPosition_pub = rospy.Publisher(name + '/mavros/setpoint_position/global', GlobalPositionTarget, queue_size=100)
28          self.setTargetPositionGlobal_pub = rospy.Publisher(name + '/mavros/setpoint_raw/target_global', GlobalPositionTarget, queue_size=100)
29          self.setTargetRawPositionGlobal_pub = rospy.Publisher(name + '/mavros/setpoint_raw/global', GlobalPositionTarget, queue_size=100)
30
31          self.setNED_pub = rospy.Publisher(name + '/mavros/setpoint_velocity/cmd_vel', TwistStamped, queue_size=100)
32          self.sendNED_pub = rospy.Publisher(name + '/mavros/setpoint_raw/local', PositionTarget, queue_size=100)
33
34          self.globalLoc_sub = rospy.Subscriber(name + '/mavros/global_position/global'
35          self.mode = rospy.Subscriber(name + '/mavros/state', State, self.stateChangeC
36
37          self.position = NavSatFix()
38          self.state = State()
39          self.rate = rospy.Rate(30)
40          self.mission = Mission()
```

Reference: src/domain/Vehicule.py:261

Figure 44 – Init function of Vehicle.py

The next method, ***go_to*** is the method responsible for making the drone fly to a specific point. A snippet can be found in Figure 49.

It is a good example of topic usage as a publisher and subscriber. The coordinates are sent, by writing to "/Mavros/setpoint_raw/global ", the topic takes a *GlobalPositionTarget* object type described in 2.5.7.1.2. The state is checked, by subscribing the topic "/mavros/state", to ensure the mode has not been changed. The mode can be changed by the remote controller. The method's parameters correspond to the GPS coordinate, latitude, longitude and altitude. They are used for creating a *GlobalPositionTarget*, a data type that is required by the topic. The header attribute, in line 149, is used for transmitting timestamp data as mentioned in the Mavros documentation, "Standard metadata for higher-level stamped data types.This is generally used to communicate timestamped data in a particular coordinate frame." [54] There are two loops, the first at line 155 and the second at line 160. The former checks if the mode within the drone is correct and has not been changed. It will ensure that if the mode changes, the computation will stop and no further command will be sent. But also to ensure that the coordinate is published only if the drone is in the right mode. The second loop checks if the drone has reached its target. The Vehicle class is subscribing to the global position topic, in the *init* method. This topic is sharing the current location of the drone every time it changes. The callback function will save the current location in the attribute where it will be used to check the distance between the current and the target position. *Rospy.sleep* is an instruction, in line 161, that will allow the node to sleep for a given period. The sleep is used to prevent the node from checking the distance too often thus leading to an overload.

```
142     #toDo check altitude
143     def go_to(self, lat, lng, altitude):
144         rospy.loginfo(self.name + " is trying to go to " + str(lat) + " " + str(lng) + " " + str(altitude))
145         gp = GlobalPositionTarget()
146         gp.coordinate_frame = GlobalPositionTarget.FRAME_GLOBAL_REL_ALT
147         gp.type_mask = 4088
148         gp.latitude = lat
149         gp.longitude = lng
150         gp.altitude = altitude
151         gp.header.stamp = rospy.Time.now()
152
153
154         target = {}
155         target['latitude'] = lat
156         target['longitude'] = lng
157
158         while(self.state.mode == 'GUIDED'):
159
160             self.setTargetRawPositionGlobal_pub.publish(gp)
161
162             # 2 because of the accuracy problem that Aero's
163             while(Util.get_distance_meters(self.position, target) > 2.0):
164                 rospy.loginfo(self.name + "is flying to target" )
165                 print self.name + " remaining to target : " + str(Util.get_distance_meters(self.position, target))
166                 rospy.sleep(0.2)
167
168
169             rospy.loginfo(self.name + "Reached the point : " + str(lat) + " "+ str(lng))
170             return
171
172         rospy.loginfo("Oups we cannot fly there because mode has changed to : " + self.state.mode)
```

Figure 45 – Go_to function

Single drone flight : Mission and normal flight

An example of usage of the Vehicle class can be found in the uav1.py file, illustrated in Figure 46. The threads are not mandatory, if it is for one drone flight. The solution without thread has been tested successfully. However, multiple drone flights require threading. Each drone needs to have its thread to call functions. The class has two methods, launchMission and launchGoTo. The former launches a mission, the latter launches a flight using the guided mode. Both flight three coordinates before returning to the home location. There are several ways to launch a Mission, as described in section 2.2.3.5. In this example, the Mission starts first by setting the Mission object of the vehicle, in line 9, and 10, and push it to the autopilot's memory. Afterwards it is armed and asked for takeoff. The first command is mandatory while the second is not. When pushing a waypoint list to the autopilot, each waypoint can be associated with a command like take-off to be executed by the autopilot.

Waypoint *:*

In Figure 47, one can find a representation of a waypoint. Waypoint is only used within Mission mode. The frame represents the coordinate system used to guide the drone. As mentioned, a command can be added to a waypoint as shown by the enumeration MAV_CMD in the figure. One can select the current waypoint with *is_current*. Waypoints are flown automatically one by one, in the order. This can be disabled by the *autocontinue* attribute.

Param1 is the hold time in seconds, at the waypoint. Param2 is the acceptance radius (the waypoint is accounted as reached when the drone penetrates the radius). Param3 is the "pass by" value in meters. If it is 0 the drone will pass through the waypoint. Otherwise it will pass

by depending on the value in meters. Param4 is the yaw angle desired. The last three attributes, x_lat, y_long and z_alt are the coordinates of the target.

At line 35, of the snippet in Figure 48, PrepareToFlyAndTakeoff method prepares the UAV to fly and orders a takeoff. The method is changing the mode to "guided" (line 133), arms the drone (line 135) and takeoff to a given altitude (line 137). Between each command, a sleep is required to let the drone execute the command.
Go_to_m (m stands for multiple) is a method that uses "go_to". It takes a list of coordinates as parameters to loop (line 210) on it and call go_to for each point. Both methods are illustrated at Figure 48 and Figure 49.

```
# see enum MAV_FRAME
uint8 frame
uint8 FRAME_GLOBAL = 0
uint8 FRAME_LOCAL_NED = 1
uint8 FRAME_MISSION = 2
uint8 FRAME_GLOBAL_REL_ALT = 3
uint8 FRAME_LOCAL_ENU = 4

# see enum MAV_CMD
uint16 command
uint16 NAV_WAYPOINT = 16
uint16 NAV_LOITER_UNLIM = 17
uint16 NAV_LOITER_TURNS = 18
uint16 NAV_LOITER_TIME = 19
uint16 NAV_RETURN_TO_LAUNCH = 20
uint16 NAV_LAND = 21
uint16 NAV_TAKEOFF = 22
# TODO: ROI mode

bool is_current
bool autocontinue
# meaning of this params described in enum MAV_CMD
float32 param1
float32 param2
float32 param3
float32 param4
float64 x_lat
float64 y_long
float64 z_alt
```

Figure 47 - Waypoint structure.

```python
#!/usr/bin/env python

import rospy,time
from domain.Vehicule import Vehicule
import threading


def launchMission():
    vehicule.setDummyMission()
    vehicule.launchMission()
    vehicule.setMode('GUIDED')
    time.sleep(2)
    vehicule.arm(True)
    time.sleep(2)
    vehicule.takeOff(18)
    time.sleep(4)
    vehicule.setMode('AUTO')
    rospy.spin()

def launchGoTo():
    points = []

    p1 = {'latitude': 41.714679, 'longitude': -86.242182, 'altitude': 17}
    p2 = {'latitude': 41.714801, 'longitude': -86.241812, 'altitude': 19}
    p3 = {'latitude': 41.714786, 'longitude': -86.241171, 'altitude': 22}

    #Altitude test
    # p1 = {'latitude': 41.714679, 'longitude': -86.242182, 'altitude': 8}
    # p2 = {'latitude': 41.714679, 'longitude': -86.242182, 'altitude': 12}
    #p3 = {'latitude': 41.714679, 'longitude': -86.242182, 'altitude': 15}

    points.append(p1)
    points.append(p2)
    points.append(p3)
    vehicule.prepareToFlyAndTakeOff(10)
    vehicule.go_to_m(points)
    vehicule.RTL()

if __name__ == "__main__":
    rospy.init_node('SingleDroneFlight', anonymous=True, log_level=rospy.INFO)
    vehicule = Vehicule('')
    t1 = threading.Thread(target=launchGoTo)
    t1.start()
    t1.join()

    rospy.spin()
```

Figure 46 - Client class, uav1.py

```python
    #Will change the mode of the UAV, arm and takeOff to the right altitude
    def prepareToFlyAndTakeOff(self, altitude):
        self.setMode_srv(0, 'GUIDED')
        rospy.sleep(1)
        self.arm(True)
        rospy.sleep(2)
        self.takeOff(altitude)
        #self.go_to_altitude(altitude)
        rospy.sleep(15)
```

Figure 48 - prepareToFlyAndTakeOff method

```python
    def go_to_m(self, points):
        for x in range(len(points)):
            self.go_to(points[x]['latitude'], points[x]['longitude'], points[x]['altitude'])
```

Figure 49 - go_to_m method.

Critics:

1. The vehicle class should handle the thread within its class instead of declaring and launching the thread in the client class. Therefore the client class has fewer manipulations to do.
2. Second, when a command is sent, the methods don't handle failure and error messages. Methods should handle errors and throw an exception so that the next commands are not executed. In the current system, the methods are only showing an error message to notify the failure but they will not block the execution flow. However, for each feature the state of the drone is checked. For instance, before take-off an instruction ensures that the drone is armed. Otherwise it will not work.
3. Mode name in "go_to" function could be declared as constant instead of being hard-coded.
4. Altitude is not checked for one reason : there were inconsistent data between MavProxy and MAVROS. MAVProxy was giving an altitude while MAVROS was giving another value, therefore it was impossible for me to check the altitude correctly. I later found that the topic that gives the location of the drone ("Mavros/global_position/global", which I used for checking the altitude) is using the MSL altitude [47] while MAVProxy is using the relative altitude. The relative altitude is given by another topic called "mavros/global_position/rel_alt".

Figure 50 illustrates the interactions between the node SingleDroneFlight (uav1.py) and MAVROS. The name of the node is set and declared in line 40 of Figure 46. The second parameter, anonymous, ensures a unique name for the node, in case many of them are launched. There are two important topics, /Mavros/setpoint_raw/global " to publish target point and "mavros/global_position/global" to get the current location of the drone (used to determine if the target is reached). The graph shows that SingleDroneFlight also publishes to other topics such as "/Mavros/setpoint_position/global" however it is not the case. Vehicle class has registered to publish on these topics but does not publish. These have been used for experiments but are not used.

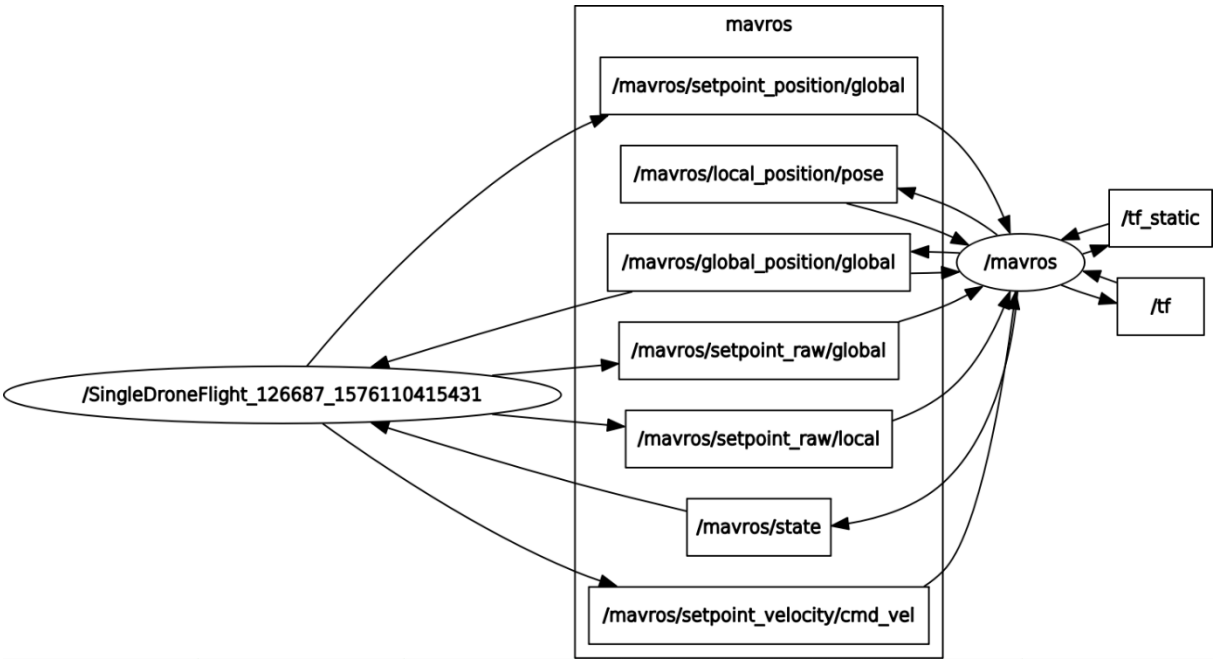

Figure 50 - Graph representation of the network, for a single flight.

3.4.2.2 Difference between PX4 and Ardupilot

There are differences in the implementation between PX4 and Ardupilot. The main difference is located in the sequencing of the events. A certain number of events need to be triggered in a particular order. These events are related to switching modes, sending coordinates, ordering a take-off and landing command. These two state diagrams illustrate the differences between the two autopilots. Although they seem to be similar, there are subtle differences with major outcome. The major variance is located before switching the mode to Offboard. With PX4 before switching to Offboard mode, a stream of points needs to be sent before changing mode. For that reason, an arbitrary (it could be less or more) number of 100 points are being published before switching the mode to Offboard.

3.4.2.3 Messages passing during flight

The second step of development is the ability to exchange messages between two vehicles while they are flying. Launching two drones doesn't require additional code, it is the same instruction as launching one drone, appearing twice. Each drone having its own instructions.
The goal is to show that it is possible to exchange information while flying by a proof of concept. I've decided to implement a " master and slave" flight to achieve the goal. The idea is to send positions from one drone to another, the position is the data exchanged. The current position of the Master is shared through Topics where the slave will subscribe. The master will receive a route to follow and the slave will have to follow the master without knowing the route in advance. Once the master is starting to move, the slave will be notified and the

positions of the Master will be transmitted to the Topic "/setpoint_raw/global" responsible for guiding the drone. The slave will then fly the same path as the master.

Critics:

1. ***Missing synchronization***: In the current solution, there is no synchronization between the two UAVs. It means that the follower does not wait for the Master to move to follow it. The follower will immediately follow the master that leads to a safety issue. While the Master takes off, the slave can also takeoff and immediately join the Master at its location. Another example is during the return to home. If both of them return at their initial position at the same time, they might crash during the flight as the takeoff position can sometimes be close to each other's. Moreover, each of the drone needs to have its own altitude, otherwise two drones flying at the same altitude may crash.
   In the current state, the solution is implemented with a delay before triggering the follow process. This will allow the master to have enough time to take off and start its route before the slave to start its mission.

Possible solutions:

There are several ways to address this issue. One of them is to create states for each state of the flight. These states can be taking off, altitude reached, flying, destination reached (intermediary or final point), hovering and landing, waiting for "follow" signal, waiting for instructions, etc. Some of these states can already be found in the state topic of Mavros, "Mavros/state". However, it is limited to : connected, armed, guided and the mode name. Therefore, additional states should be created and shared though a topic and adapt the behavior of the slave in consequence via the callback function. This way a synchronization can be implemented between the drones. For instance, when ordering a landing every drone will wait until ever other participant is at its home location, before landing.

Figure 52 illustrates the nodes and topics used when two drones are flying. The graph is the same as a single drone flight. The idea is the same, the main difference is located on how the target points are given to the second UAV. These points are taken from the position of the master, and given to the follower. This part is achieved with the methods in Figure 51. *Follow* method subscribes to the Master's position, the parameter *name* corresponds to the master name. Each drone has its own Mavros with as namespace the name of the drone, as shown in Figure 52. Thus with the parameter the slave can subscribe to the master's location. The callback function will be called, every time the Master's position changes, where the data (location of master) will be given as target location to the Slave.

```
44      #Subscribe to master's position
45      #Then as soon as the master starts moving
46      #The slave is going to follow
47      def follow(self, name):
48          self.globalLocOf_sub = rospy.Subscriber(name + '/mavros/global_position/global', NavSatFix, self.globalLocationOfCallback)
49
50
51      #Altitude hardcoded to ensure safety,
52      #global position used MSL altitude
53      def globalLocationOfCallback(self, data):
54          self.go_to(data.latitude, data.longitude, 16)
```

Figure 51 - follow method and callback associated.

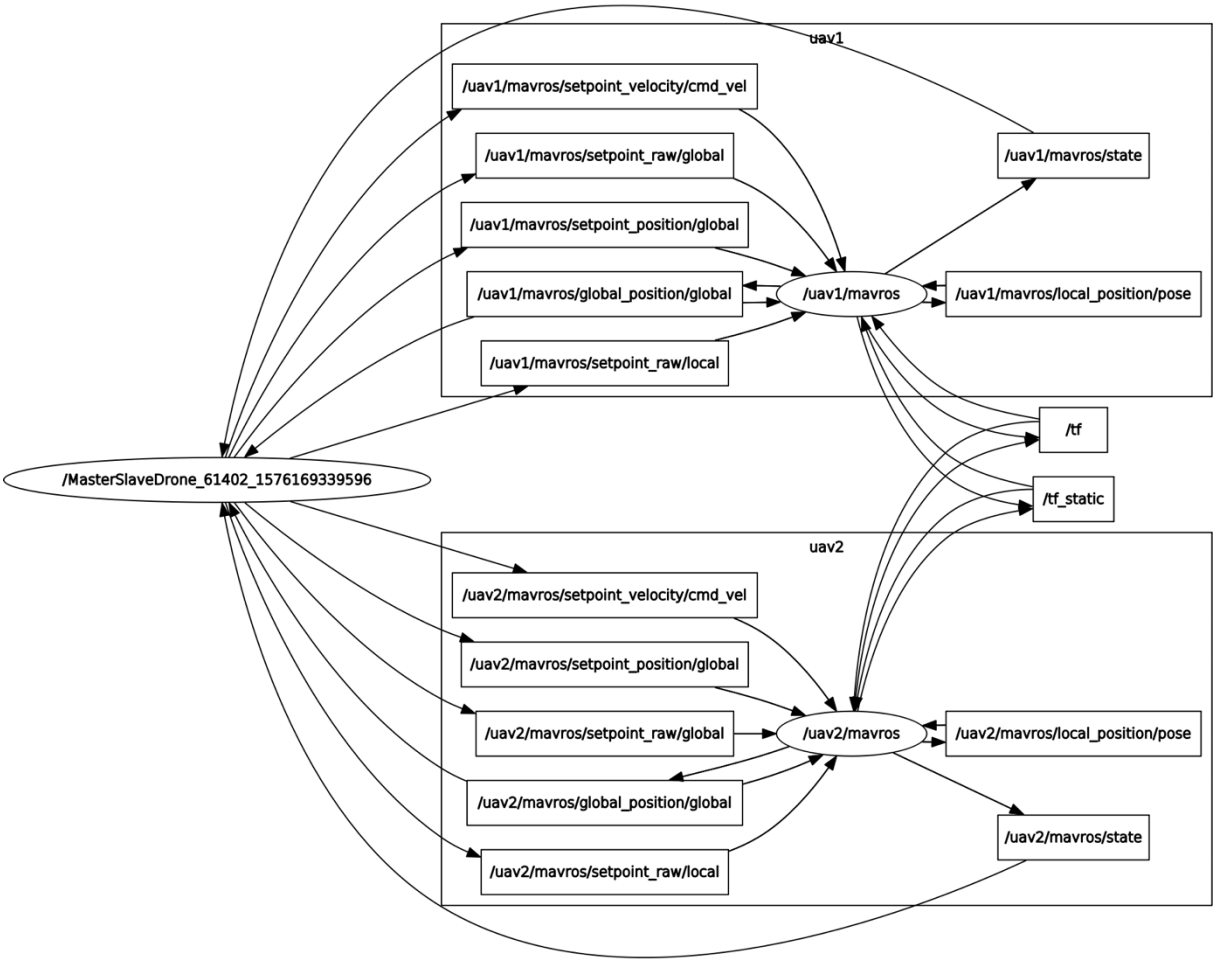

Figure 52 - Graph representation of Master and Slave node.

3.4.3 Test cases

The tests written are all functional tests. They are focused on the effect that the code has on the drones. The tests were all scenarios executed to ensure that the behavior observed is correct. A test is considered as successful if the drone responds to commands successfully and fly the given locations. A command is successfully executed if it takes effect and no error messages is returned.

The scenario for each test is to make the drones takeoff, fly several coordinates and end by a return to initial takeoff position and land. All the tests have been done inside the simulator, unless mentioned otherwise. The simulator used for the test is Ardupilot SITL and MAVProxy. The outdoor test are done in a field used for this purpose (Figure 53). The tests are incremental, starting with simple interaction before becoming more elaborate.

Figure 53 - Outdoor test setup.

Simple command : arm & takeoff

The first iteration was a simple script that changes the mode to "guided", arms the drone and takeoff to a given altitude before landing it at the same place. The goal was to get familiar with Topics, Services and the simulator environment.

Flight 1 : Mission mode

Then I have tried a flight using Mission mode, by giving three locations (hardcoded) to fly. The reason why I first started with Mission mode is because it is easier to fly as a lot of steps are automated. Such as the takeoff, landing, the succession of the waypoint to fly, and more importantly check if a target location is reached. Thus Mission flight requires less effort than a normal flight. An example can be found in Figure 46.

Flight 2 : Guided mode : Single drone

After successfully passing this step, the next step was flying the drone in Guided mode. It requires more management of the different steps of a flight but also more freedom compared to Mission mode. The test executed a simple scenario. The drone takes off and flies to three given points and finishes by coming back home with the RTL mode and land. Additionally to

the simulator, this test has also been achieved outdoor with success. Table 3 contains the data of the flight, including the location to fly and the location flown. The latitude and longitude flew by the UAV are slightly different from the coordinate given because of two reasons. The distance between the actual position of the UAV and the target is done via calculation thus introduces approximation. And second the acceptance radius around the point is one meter. A target is considered as achieved when the drone is as close as one meter to the target. The evolution of the coordinates are illustrated in Figure 55. The three lines are an approximation where the target locations are flown. Each waypoint has its own altitude on purpose, to observe the altitude behavior. The elevation variation can be observed in the last graphic where it starts with the take off at nine meters above ground. After the takeoff we can see an increase of the altitude as the UAV is heading to its first target with a different altitude.
The path flown is illustrated in Figure 54. The yellow and blue dotted lines of the flight represent the mode in which the drone flew, which are guided and RTL mode. Each dot represents a point where the drone flew, it is composed of 325 points.

Table 3 – Flight location data

Location to fly			Location flew		
Latitude	Longitude	Altitude (m)	Latitude	Longitude	Altitude (m)
41.714679	-86.242182	17	41.7146789	-86.2421819	16,99
41.714801	-86.241812	19	41.7148009	-86.2418122	19,01
41.714786	-86.241171	22	41.714786	-86.2411737	22,01

Figure 54 - Actual flight path

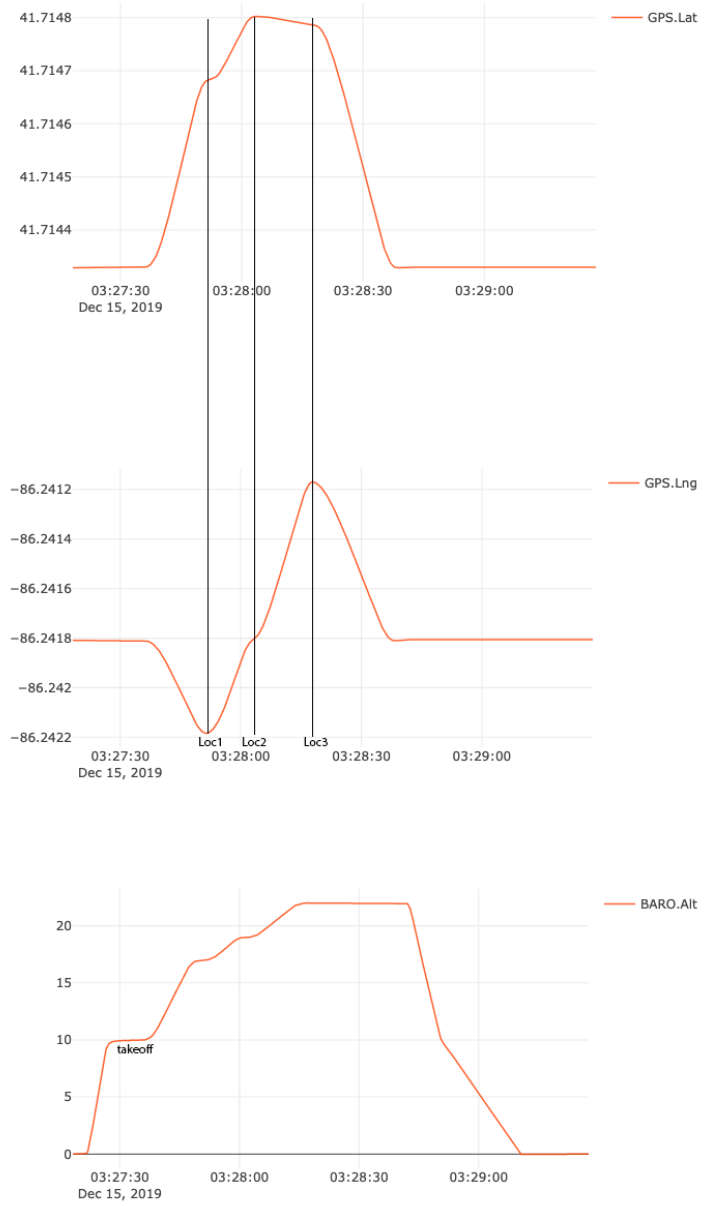

Figure 55 - Latitude, longitude, altitude data.

Flight 3 : Multiple drones

The next increment is a multi-drone flight, first with two drones. This step involves introducing threads to make multiple drones fly. So I've first launched the previous test with threads which has been a success. The scenario for multiple drones is the same as a single drone except the number of drones flying. The target location is different each of the drone has a specific path to fly. It has been tested in two different environments, in the simulator and outdoor. The simulator test was a success. The outdoor test, however, did not work at first because of a configuration problem on the drone itself. The port of the Autopilot (of the drone), given to

Mavros was not correct. After correcting it, the second try was successful. Each of the drones had six points to fly and finished by hovering at the last point. The return was handled manually because no synchronization had been implemented as mentioned in the critics. As the test was successful, the next step was the communication between two drones while flying.

Flight 4 : Master and Slave

This test is using two drones, a leader where target points are given and the follower. The idea was to make the leader fly three locations and the follower could follow without knowing the locations. Instead, it subscribes to leader position and follows the same path. The return is also executed manually to ensure safety. The Master and Follower do not fly at the same altitude. The former flies twelve meters while the latter at sixteen meters. The path flown is the same as Flight1, however, the behavior is not.

Issue

When the Master arrives to a target, it is somehow stuck on the target point. The problem is coming from the loop that checks the distance between the location of the drone and the target (in Figure 45). While the drone is moving toward the destination, the remaining distance stays the same for a little time of period (4 loops). The current position does not seem to be modified. While the drone is moving forward, the remaining distance does not refresh properly. The value decreases slowly until matching the condition. A snippet of logs output are available in Figure 56, the loop is run four times before the value changes which are different from the normal behavior. It results that the drone is hovering at the target location, while being stuck in the loop that calculates the remaining distance. As the loop running, it prevents the publication of a new point. The incident affects the latitude and longitude graph, in Figure 57. There are 2 clear horizontal lines, starting at 112 seconds, where the value does not change. These correspond to the target point where the UAV is hovering. The first and last horizontal lines are coming from the takeoff and hovering above the last target.
The same code is used with the previous flight, with two drones and with three, the incident did not occur. I tried to search the origin but I could not find it.

Hypothesis of the Origin

I suspect that it is linked to the threads. Indeed, the Python Interpreter limits the multithreading principle by using a Global Interpreter Lock (GIL). The GIL ensures a safe access to Python Interpreter by putting a lock, thus ensures that only a single thread can access to the interpreter. A thread cannot execute Opcodes (low-level operations) without the GIL. In the current solution, each thread has its own Vehicle object. Inside the Vehicle class, the *go_to* method is using a method of the Util class. The method is named *get_distance_meters*. While

the drone is flying, each thread is running the loop that checks the distance. Thus each thread is trying to access the same instruction, which could lead to concurrence between threads.
The reader might wonder, why is the Flight4 situation any different from the previous flights or the upcoming flights ?
When a thread is executing the *go_to* second loop (Figure 45, line 164) that checks the remaining distance, the thread sleeps for a small among of time, 0.2 seconds in the example (Figure 45, line 167).
In Flight4, the slave receives the position of the Master every time it changes. Thus the distance between two points is so small that the condition of the loop is always false (condition : *distance > 2 meter*s). This instruction rospy.sleep, responsible for sleeping the thread is never executed, while in the other flights, this instruction is executed which release the lock. Moreover, the *go_to* method is called every time the position of the Master is changed, which involves a huge amount of calls. This could possibility impeach the release of the lock because the execution of the *go_to* method is constantly running.
However, these are assumptions. I did not have enough time to check and verify my hypothesis.

Alternative flight

Since the problem is coming from the calculation of the remaining distance, I decided to use the Mission mode for the Master. The calculation is handled by the Autopilot. It determines when a target is reached. The incident did not occur in Mission mode. The graph, in Figure 59, proves it. It looks slightly different from Flight2 because an arbitrary time of 2 seconds of hovering above each waypoint has been set. And also because there is no RTL instruction.

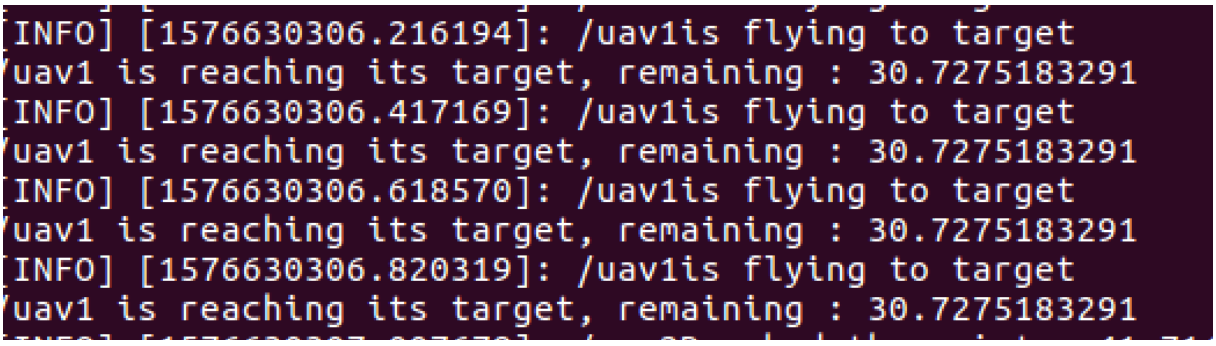

Figure 56 – Flight4 - UAV1 remaining distance log.

Master and Follower data comparison

While targets and the points flown are the same, the Follower takes more time than the Master to complete the same path. This is completely normal as the Follower receives the position to fly one by one as the Master moves. The Follower has more points to fly, each of them being very close to each other. The process thus takes more time.

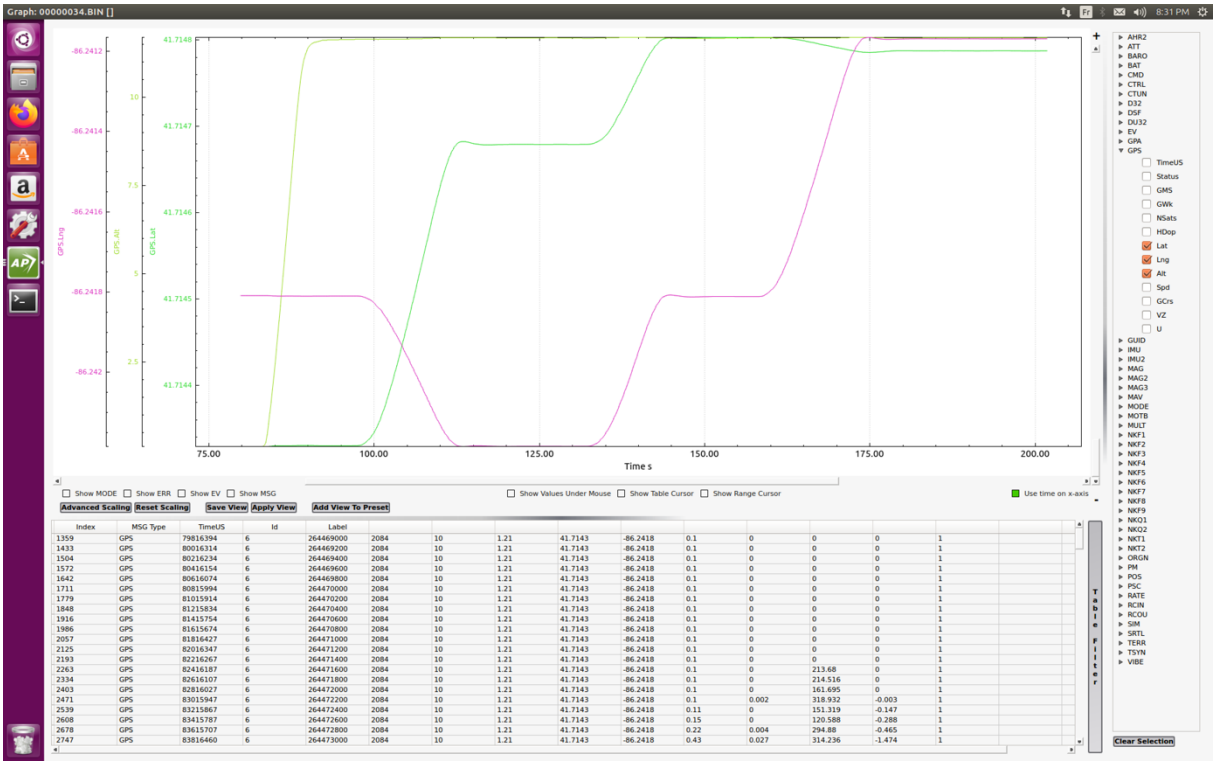

Figure 57 - Master - Guided mode

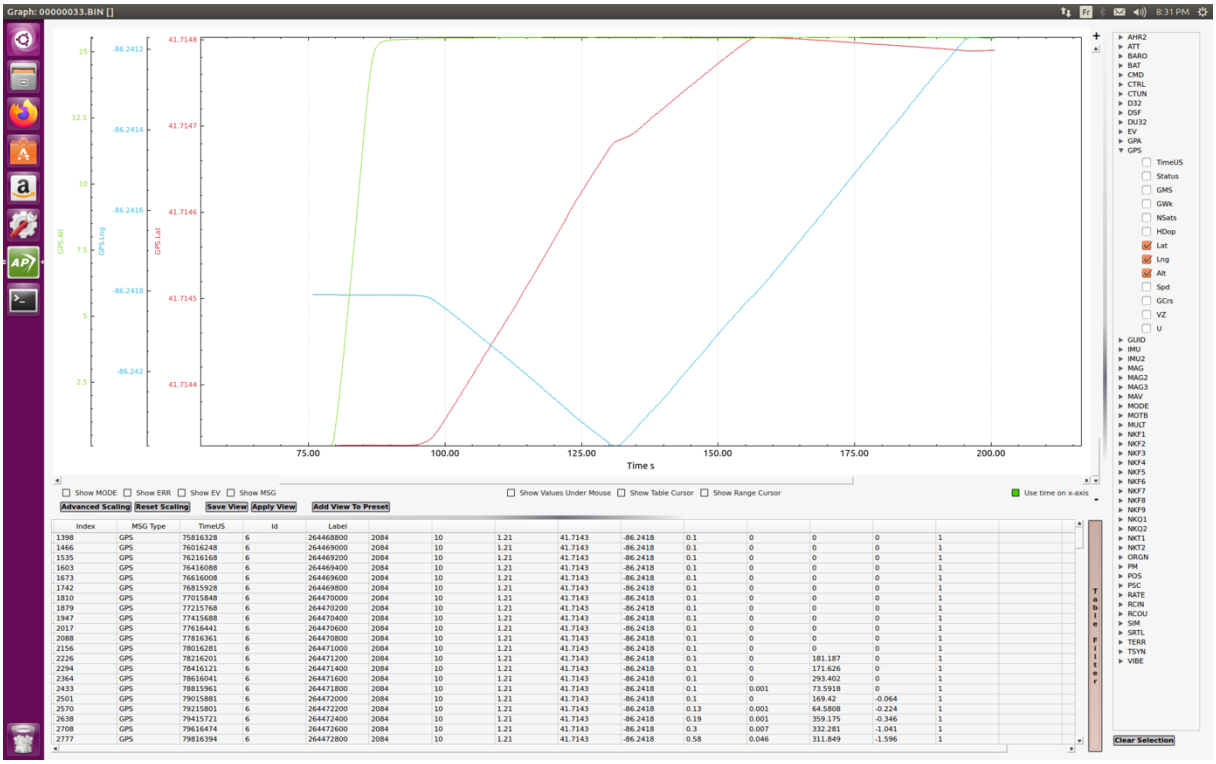

Figure 58 - Follower - Guided mode

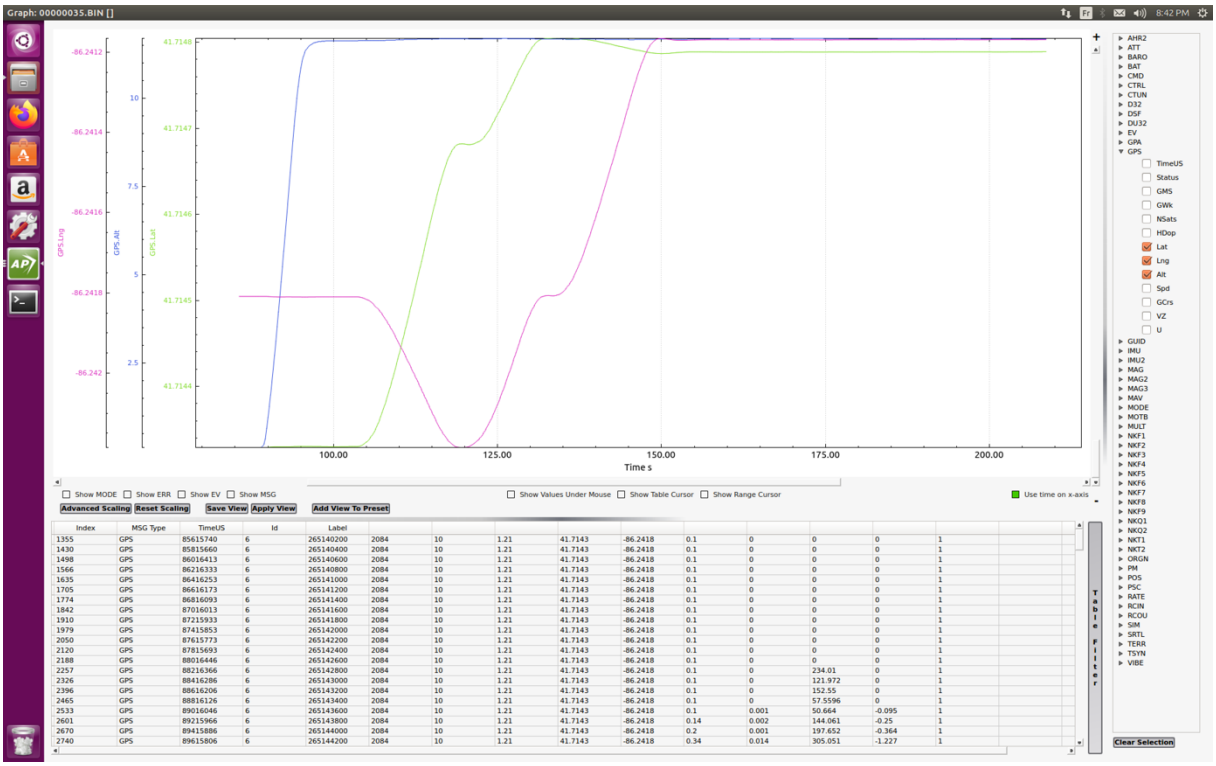

Figure 59 - Master -Mission mode

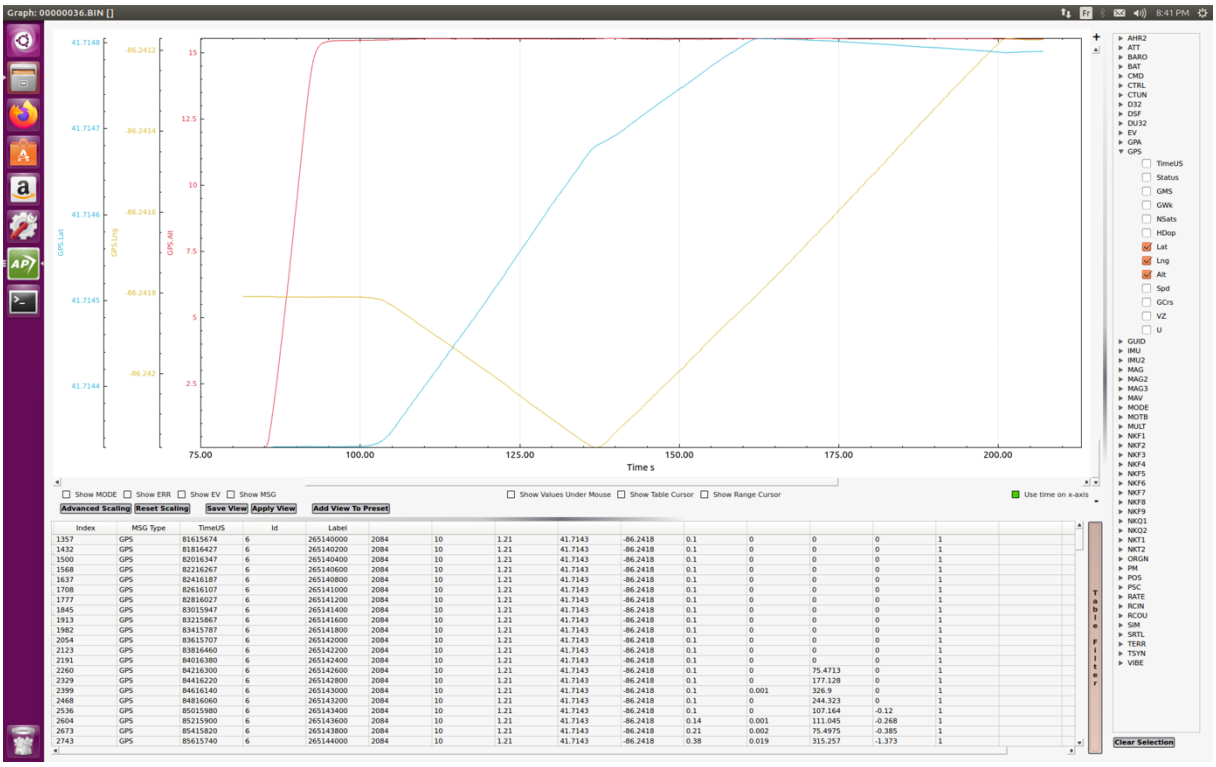

Figure 60 - Follower - Mission mode

Flight 5 : Triple drones with JSON file.

The next test case was a real scenario test by taking the result of a routing algorithm and give it to ROS as input. The algorithm had been developed for a search and rescue project using drones, by the team of Prof Cleland-Huang. The algorithm takes several parameters based on a goal model. These parameters can be the number of drones, battery consumption, time constraints, priority areas, etc. A priority area can be a river bank or a zone inside the river. Based on the goals, the algorithm determines route for each drone to fly. Each drone flies area determined by the algorithm. For this test a fake river has been set in the middle of the test field. The objective of the test was to determine if ROS could play the same role as Dronekit within the project as Dronekit was used to fly the drone. I wrote a node that takes the file as input, parses to determine how many drones need to be launched, then for each drone assign the route that needs to be flown. Each drone can have several route assignments. The test inside the simulator performed well, the path of one drone can be seen in Figure 61. The drone flew across the fake river from one bank. The other two paths can be found in Figure 62 and Figure 63.

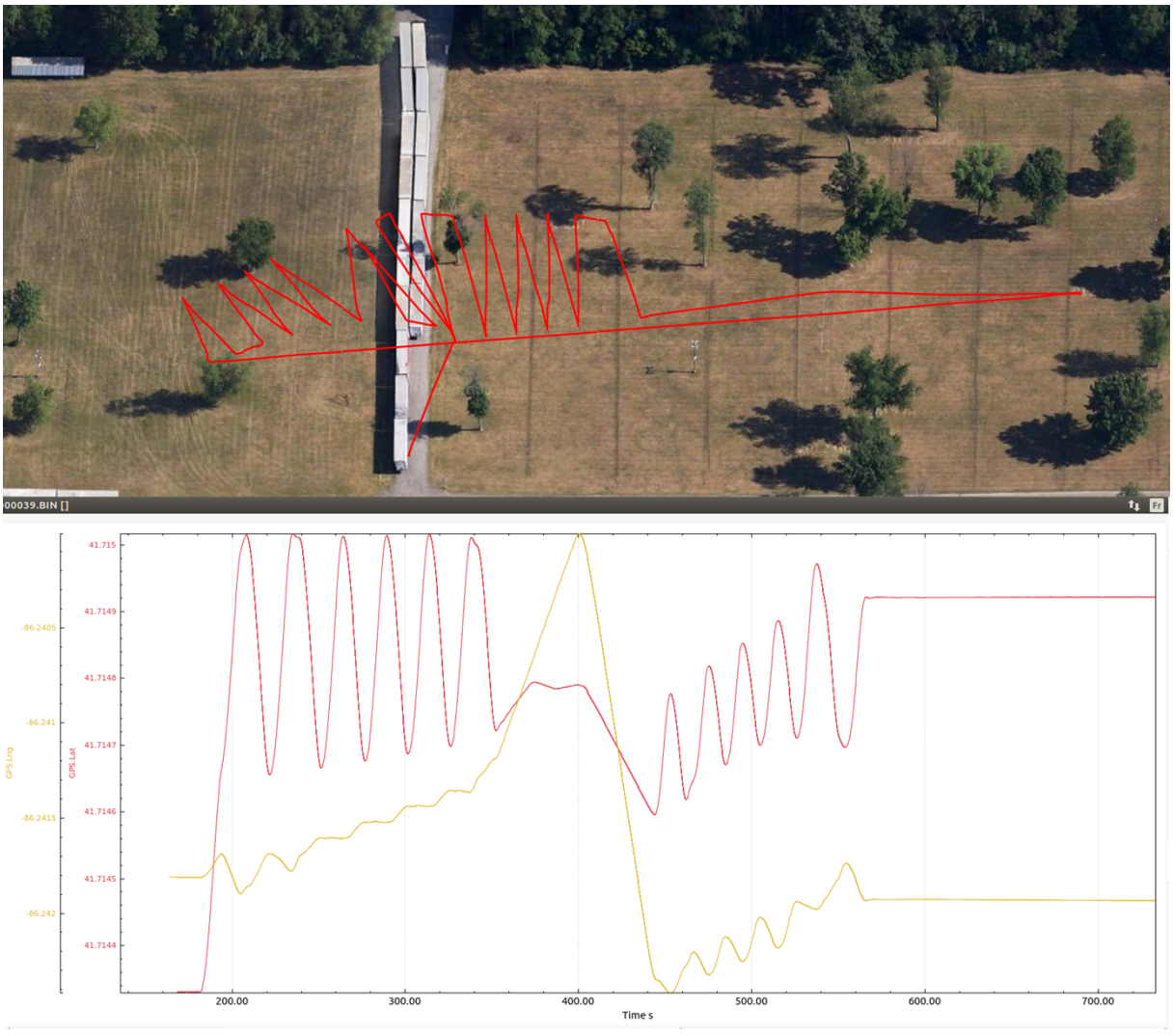

Figure 61 - Flight5 - UAV1 - Path and flight data

However the outdoor test did not succeed. The test is using three drones, two of them taken for the test were not the same used for the previous tests.
Therefore, before launching the actual test, I first launched a previous test that passed with success. to ensure that the drones were properly configured and working.
Then two attempts have been made with three drones. On both of them, one of the drones refused to take off. The problem came from the configuration of the three drones within the network. There were collisions or confusion inside the network, the MAVLink messages sent from the computer being not correctly routed. So it didn't reach the destination or reached the wrong one. This is probably a configuration issue with the messages being sent from Mavros to the right FCU. This test was the last one and it happened one week before my departure. Therefore I wasn't able to launch another test by lack of time and due to weather conditions.

Unfortunately, I did not extract available log files from the drone due to the low number of outdoor tests, the fact that drones were reset several times between flights and the usage of different drones. These are the reasons why I did not publish quantitative data for an outdoor test. Therefore I cannot compare the outdoor and simulated flight. However these previous tests are enough to establish that ROS is indeed capable of making drones fly. The drone responds well to command and follow the instruction that is being given according to the result of Flight1 and Flight3.

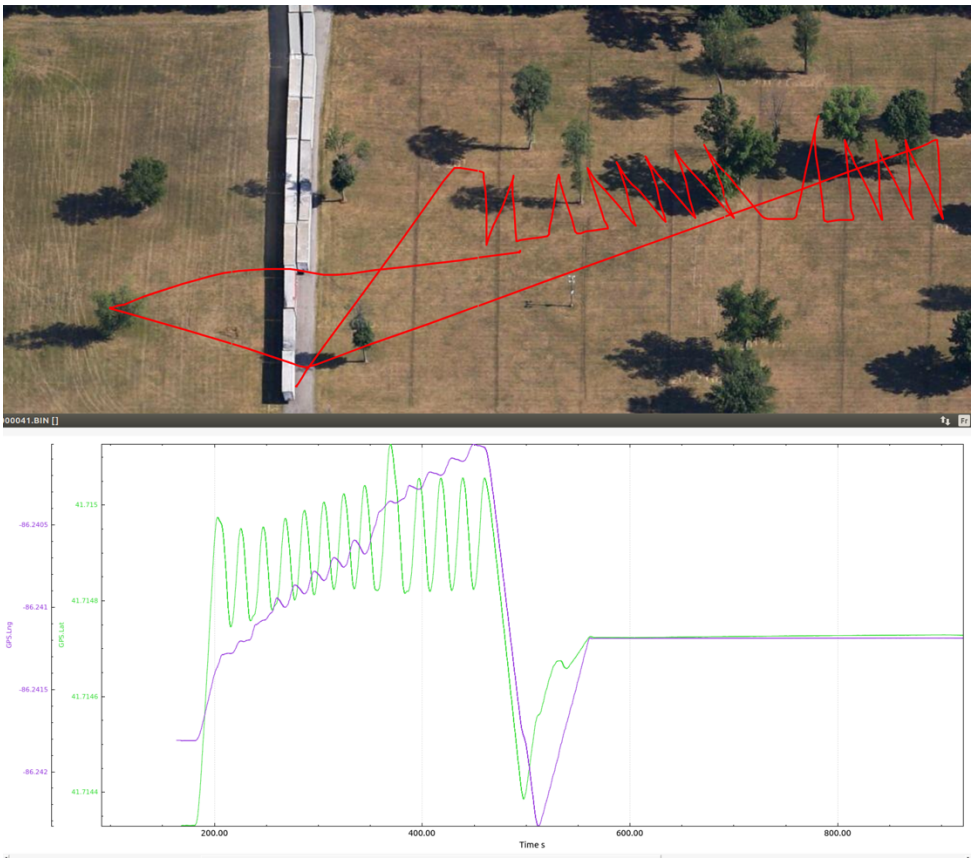

Figure 62 – UAV2 - Path and flight data

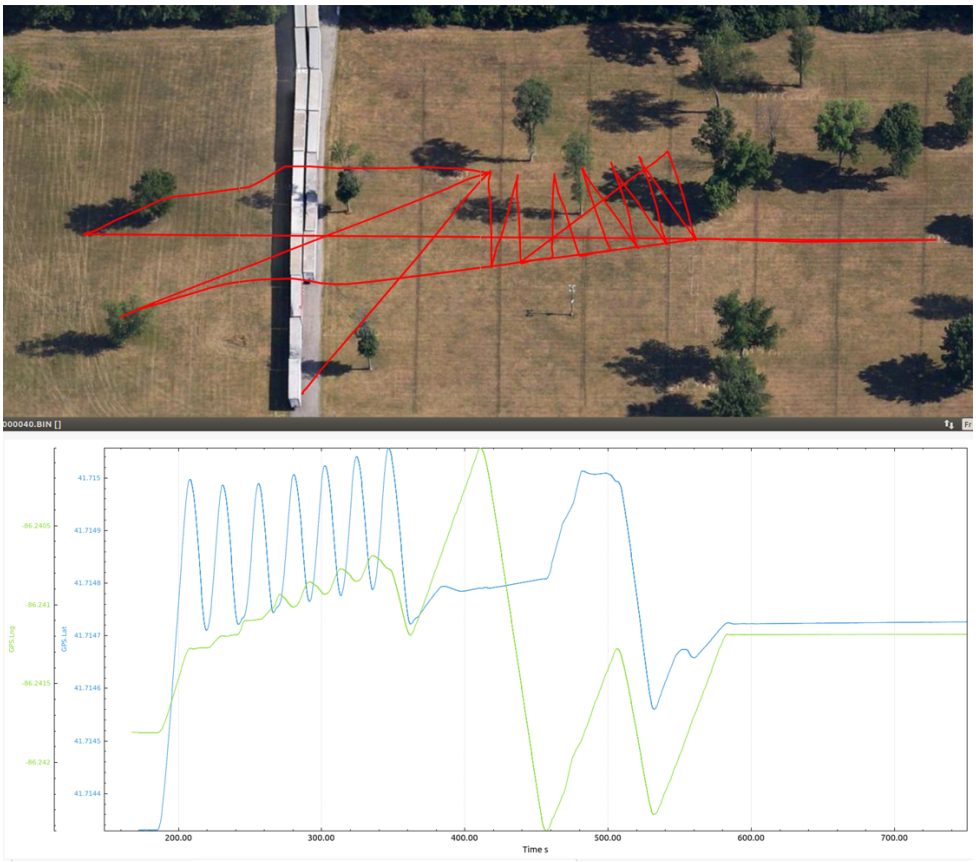

Figure 63 - UAV3 - Path and flight data.

Chapter 4 Conclusion

In the past few years, the interest for UAVs has tremendously increased, first through military and then civilian applications, including surveillance, mapping, search and rescue missions. The technology has evolved to be suitable for civilian usage, with improvements in terms of cost, size, design and battery range. The small UAVs have become light-weight, making them transportable and even launched by hand. The progress will not stop anytime soon as the usage of UAV has just started this decade. The UAVs are technology that did not come to maturity yet and therefore will grow before becoming used in daily life.

This document has the intent of introducing the Unmanned Aerial Vehicle and present a proof of concept of a drone controller using Robotic Operating System, a robot oriented framework. After spending three months, in Prof. Cleland-Huang's team at the University of Notre Dame, Indiana.

It began with an introduction to UAVs in general, with its origin and the different terms used to describe them. Afterwards, I gave an explanation of the different existent classifications used for military but also a general classification regarding the shape of an UAV it was followed by an explanation on how a UAVs move introducing the navigation axes, called Roll, Pitch and Yaw, with their definitions and effects on the aircraft's body as well as the different systems of coordinates. The next section introduced components that are part of the UAV such as the communication protocols, Autopilot and control software UAVs.

Different elements of communication to exchange data, command and a messaging protocol have been presented : Telemetry to exchange the data, Radio Command to send control commands and MAVLink protocol to exchange information.

Once the communication protocols are discovered, both parties need to be defined.

The discussion is established between a software on the ground and a component on the drone itself. This latter is carried by the Autopilot, such as Ardupilot and PX4. It is responsible for flying the drone and responds to commands sent from the ground. It is also providing flight mode to assist the user during the flight. A state chart of a flight has been explored to identify the different states of the UAV during a flight.

The commands sent from the ground can be carried by either software like MissionPlanner, QGroundControl or software written using framework like Dronekit or ROS. Both have been also been introduced. To ensure the safety of the software written by programmers, we have also explored simulators such as Ardupilot and PX4, both being the most popular simulators.

Finally, this work had two objectives : (1) Is ROS suitable for drone. (2) How to achieve data exchange between flying drones? Both of these objectives have been addressed in the last chapter contribution. It started by providing an explanation of a proof-of-concept aimed to control a drone, written with ROS followed by an explanation of the architecture and the role of each classes as well as the different flights executed to test it. This aimed to determine if

ROS can be used to control drones, like Dronekit which is specially designed for this purpose. The data extracted from each flight helped us to conclude that ROS, despite being a framework for robots, can be used to control UAVs thus respond to the first objective. Moreover, we have also seen that indeed messages can be exchanged during flight with Flight5, which respond to the second objective. The proof-of-concept is an experimentation and is not complete nor safe to fly drones other than simple flight routes. However, I am aware of the limitations and address some of them with possible solutions. Some of them are the errors that are not catched and properly handled or the limitation of Python with multi-threaded executions.

The drone universe is a wide domain of study, it goes from design study, study of trajectory, path planning for a swarm of UAVs, military and civilian surveillance, etc. This document is an attempt to introduce a general view of the most important elements that compose an UAV and its elements around. It is also a first approach and experimentation to the drone software development using ROS. As consequence, a lot of field studies have not been explored nor a deep analysis of components have been made in the document.

5. Bibliography

1. **Blyenburgh, Peter.** *Current situation and consideration for the way forward.* 2000.

2. **United Kingdom, Ministry of Defence.** Unmanned Aircraft Systems, Joint Doctrine Publication 0-30.2 . 2018.

3. **Francesco Nex, Fabio Remondino.** UAV for 3D mapping applications : a review. *Applied Geomatics.* 2014, Vol. 6.

4. **P. Valavanis, Kimon and J. Vachtsevano, George.** *Handbook of Unmanned Aerial Vehicles.* s.l. : Springer, 2015.

5. **Dalamagkidis, Konstantinos.** Aviation History and Unmanned Flight. [book auth.] Kimon Valavanis and George Vachtsevanos. *Handbook of Unammaned Aerial Vehicle.*

6. **The Early Days Of Drones. *War History Online.* [Online] https://www.warhistoryonline.com/military-vehicle-news/short-history-drones-part-1.html.**

7. World War 1 History: The Kettering Bug—World's First Drone. *Owlcation.* [Online] https://owlcation.com/humanities/World-War-1-History-The-Kettering-Bug-Worlds-First-Flying-Bomb.

8. WORLD WAR I: 100 YEARS LATER, Unmanned Drones Have Been Around Since World War I. *Smithsonian.* [Online] https://www.smithsonianmag.com/arts-culture/unmanned-drones-have-been-around-since-world-war-i-16055939/.

9. center, Joint Air Power Competence. *Strategic Concept of Employment for Unmanned Aircraft Systems in NATO.* 2010.

10. *Classification of Unmanned Aerial Vehicles.* Arjomandi, Dr. Maziar. Report for Mechanical Engineering class, University of Adelaide, Australia, Adelaide : s.n., 2006.

11. Gaurav , Singhal, Lini, Mathew and Babakumar, Bansod. Unmanned Aerial Vehicle classification, Applications and challenges : A Review. *Preprint.* 2018.

12. Torenbeek, Egbert. Synthesis of Subsonic Airplane Design, Chapter 8. *Airplane weight and balance.* 1982.

13. *Aircraft Maximum Gross Takeoff Weight – MGTOW.* [Online] https://www.skybrary.aero/index.php/Maximum_Take-Off_Mass_(MTOM).

14. Crouch, Collier C. *Integration of mini-UAVs at the tactical operations level : implications of operations, implementation, and information sharing .* 2005.

15. Hermes 90. *Army Recognition.* [Online] [Cited: 11 17, 2019.] https://www.armyrecognition.com/israel_israeli_army_military_equipment_uk/hermes_

90_uav_unmanned_aerial_aircraft_vehicle_system_data_sheet_information_specifications.html.

16. Aladin 450. ***Military Factory.*** **[Online] [Cited: 11 17, 2019.] https://www.militaryfactory.com/aircraft/detail.asp?aircraft_id=1440.**

17. Elbit Hermes 450 . ***Avions Militaires.*** **[Online] [Cited: 11 17, 2019.] https://www.aviationsmilitaires.net/v2/base/view/Variant/4165.html.**

18. Global Hawk. ***NASA.*** **[Online] https://www.nasa.gov/multimedia/imagegallery/image_feature_2362.html.**

19. Ghozali S. Hadi, Rivaldy Varianto , Bambang Riyanto T. , Agus Budiyono. Autonomous UAV System Development for Payload Dropping Mission. ***The Journal of Instrumentation, Automation and Systems.*** **2014, Vol. I.**

20. Edison Pignaton de Freitas, Tales Heimfarth, Ivayr Farah Netto, Carlos Eduardo Lino, Carlos Eduardo Pereira, Armando Morado Ferreira, Flávio Rech Wagner, Tony Larsson. UAV relay network to support WSN connectivity. ***International Congress on Ultra Modern Telecommunications and Control Systems.*** **2010.**

21. Douglas W. Murphy, James Cycon. Applications for mini VTOL UAV for law enforcement. ***SPIE Proceedings .*** **1999, Vol. 357, Information, and Training Technologies for Law Enforcement.**

22. ***Flight Test Results of Autonomous Fixed-Wing UAV Transitions to and from Stationary Hover.*** **Eric N. Johnson, Michael A. Turbe, Allen D. Wu Suresh K. Kannan, James C. Neidhoefer. s.l. : Collection of Technical Papers - AIAA Guidance, Navigation, and Control Conference, 2006.**

23. Henri, Eisenbeiss. ***UAV photogrammetry.*** **Zurich : s.n., 2009.**

24. Aircraft Rotation. ***National Aeronautics And Space Administration.*** **[Online] https://www.grc.nasa.gov/WWW/K-12/airplane/rotations.html.**

25. Haiyang Chao, Yongcan Cao, YangQuan Chen. Autopilots for Small Fixed-Wing Unmanned Air Vehicles: A Survey. ***International Conference on Mechatronics and Automation, Harbin.*** **2007.**

26. MISB. Motion Imagery Standards Board, ST0601.8 . ***UAS Datalink Local Set.*** **2014, Vol. ST0601.8.**

27. Mitchell M. Sisak, James S. Latimer. Frequency choice for radio telemetry: the HF vs. VHF conundrum. [book auth.] Anras ML.B., Claireaux G. Lagardère JP. ***Advances in Invertebrates and Fish Telemetry, vol 130.*** **1998.**

28. R. J. Moore, B. C. Watson, D. A. Jones and K. B. Black, C. M. Haggett, M. A. Crées and C. Richards. Towards an improved system for weather radar calibration and rainfall forecasting using raingauge data from a regional telemetry system. ***New Directions for Surface Water Modeling.*** **1989, 181.**

29. Frederick A. Voegeli, Malcolm J. Smale, Dale M. Webber, Yanko Andrade, Ronald K. O'Dor. Ultrasonic Telemetry, Tracking and Automated Monitoring Technology for Sharks. ***Environmental Biology of Fishes.*** **2001, 60.**

30. ERC Report 025. ***European Communications Office, Documentation Database.*** **[Online] https://www.ecodocdb.dk/document/593.**

31. ***International Telecommunication Union .*** **[Online] https://www.itu.int/net/ITU-R/terrestrial/faq/index-fr.html#g013.**

32. ***U.S. Government Information.*** **[Online] CFR Ch. I (10–1–02 Edition).**

33. ***ITU.*** **[Online] https://www.itu.int/net/ITU-R/terrestrial/faq/index.html.**

34. ***MAVLink.*** **[Online] https://mavlink.io/en/.**

35. Pixhawk4. ***PX4.*** **[Online] [Cited: 11 17, 2019.] https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html.**

36. QGroundControl. ***QGroundControl.*** **[Online] https://docs.qgroundcontrol.com/en/.**

37. ***Me and the rbotos.*** **[Online] https://mipsme.blogspot.com/2018/11/start-gazebo-and-px4-sitl-separately.html.**

38. ***FlightGear Flight Simulator.*** **[Online] https://www.flightgear.org.**

39. ***ArduSub.*** **[Online] http://www.ardusub.com.**

40. Dronekit. ***Dronekit.*** **[Online] http://dronekit.io.**

41. Clelang-Huang, Jane, Vierhauser, Michael and Bayley, Sean. Dronology: An Incubator for Cyber-Physical Systems Research. ***Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), 109-112.*** **2018.**

42. Dronology. ***Notre-Dame Team.*** **[Online] https://dronology.info/notre-dame-team/.**

43. Firesmith, Donald. Engineering Safety Requirements, Safety Constraints, and Safety-Critical Requirements. ***Journal of Object Technology.*** **Vols. 3, 17-42.**

44. DeLive. ***Dronology.*** **[Online] https://dronology.info/defibrillator-delivery/.**

45. ***Vaadin.*** **[Online] https://vaadin.com.**

46. Writing a Simple Publisher and Subscriber (Python). ***Robotic Operating System.*** **[Online] http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29.**

47. ROS. ***mavros_msgs/HomePosition Message .*** **[Online] http://docs.ros.org/jade/api/mavros_msgs/html/msg/HomePosition.html.**

48. rosservice. ***Robotic Operating System.*** **[Online] http://wiki.ros.org/rosservice.**

49. rostopic. ***Robotic Operating System.*** **[Online] http://wiki.ros.org/rostopic.**

50. rosnode. ***Robotic Operating System.*** **[Online] http://wiki.ros.org/rosnode.**

51. Base mode. ***Robotic Operating System.*** **[Online] http://docs.ros.org/api/mavros_msgs/html/srv/SetMode.html.**

52. ***The Tech Portal.*** **[Online] https://thetechportal.com/2016/08/17/intel-ready-to-fly-drone-aero-developers/.**

53. Intel Aero Setup. ***Github.*** **[Online] https://github.com/intel-aero/meta-intel-aero/wiki/02-Initial-Setup#flashing-the-flight-controller-rtf-only.**
54. std_msgs/Header.msg. ***Robotic Operating Sytem.*** **[Online] http://docs.ros.org/fuerte/api/std_msgs/html/msg/Header.html.**
55. M. Y. Zakaria, Moatassem M. Abdallah , M Adnan Elshafie. ***Design and Production of Small Tailless Unmanned Aerial Vehicle.***
56. Department of Defense, United States of America. ***Unmanned Aircraft System Airspace Integration Plan.***
57. ***Environmental Biology of Fishes, Ultrasonic Telemetry, Tracking and Automated Monitoring Technology for Sharks.*** **O'Dor, Frederick A. VoegeliMalcolm J. SmaleDale M. WebberYanko AndradeRonald K. 2001.**
58. Custom Mode. ***Robotic Operating System.*** **[Online] http://wiki.ros.org/mavros/CustomModes.**
59. Soares, Filomena, Garrido, Paulo and Moreira, Antonio Paulo. ***Controlo, Proceedings of the 12th Portuguese Conference on Automatic Control.*** **s.l. : Springer, 2016.**
60. Marty, Joseph A. ***Vulnerability Analysis of the MAVLink protocol for command and control of Unammand aircraft.*** **s.l. : Air Force Institude Of Technology, 2014.**
61. Stasse, Olivier. ***Robot Operating System, Introduction.*** **2016.**
62. Noda, Itsuki, et al. Simulation, Modeling, and Programming for Autonomous Robots. ***SIMPAR: International Conference on Simulation, Modeling, and Programming for Autonomous Robots .*** **2014, Vol. III.**
63. H.-P. Thamm, N. Brieger , K.-P. Neitzke , M. Meyer , R. Jansen , M. Mönninghof. SONGBIRD - an innovative UAS combining the advantages of fixed wing and multi rotor UAS. ***International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences.*** **2015, Vols. XL-1/W4.**
64. Lozano, A. Brezoescu · T. Espinoza · P. Castillo · R. Adaptive Trajectory Following for a Fixed-Wing UAV in Presence of Crosswind. ***Journal of Intelligent & Robotic Systems. .*** **2012, Vol. 69.**
65. Joao Machado Santos, David Portugal, Rui P. Rocha. An Evaluation of 2D SLAM Techniques Available in Robot Operating System. ***IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR.*** **2013.**
66. ***Research on computer vision-based for UAV autonomous landing on a ship.*** **Guili Xu, Yong Zhang, Shengyu Ji, Yuehua Cheng, Yupeng Tian. s.l. : Pattern Recognition Letters, 2009, Vol. 30.**